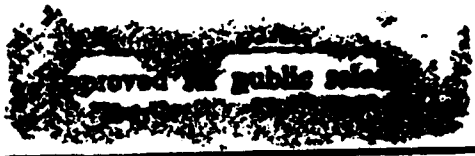
University of Southern California

Synthesis of Asynchronous Systems from Data Flow Specifications

Tzyh-Yung Wuu, USC-ISI
Sarma B. K. Vrudhula, UA-ECE

ISI/RR-93-366
December 1993

128-14



INFORMATION SCIENCES INSTITUTE

310/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

12

ISI Research Report
ISI/RR-93-366
December 1993

Synthesis of Asynchronous Systems from Data Flow Specifications

Tzyh-Yung Wuu, USC-ISI
Sarma B. K. Vrudhula, UA-ECE

ISI/RR-93-366
December 1993

This research was sponsored in part by the Advanced Research Projects Agency under contract number MDA903-92-D-0020 and in part by a grant from the National Science Foundation under award number MIP-9111206. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of ARPA, NSF, the U.S. Government, or any person or agency connected with them.

REPORT DOCUMENTATION PAGE

FORM APPROVED
OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1993, Dec.	Research Report

4. TITLE AND SUBTITLE

Synthesis of Asynchronous Systems for Dara Flow Specifications

5. FUNDING NUMBERS

MDA903-92-D-0020
MIP-9111206

6. AUTHOR(S)

Wuu, Tzyh-Yung and Vrudhula, Sarma B.K.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

USC INFORMATION SCIENCES INSTITUTE
4676 ADMIRALTY WAY
MARINA DEL REY, CA 90292-6695

8. PERFORMING ORGANIZATON REPORT NUMBER

ISI/RR-93-366

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

ARPA
3701 N. Fairfax Drive
Arlington, VA. 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A. DISTRIBUTION/AVAILABILITY STATEMENT

UNCLASSIFIED/UNLIMITED

12B. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report presents a method for automatic systhesis of asynchronous digital systems from high-level data flow specifications. We presents an extended data flow model that accurately reflects the behavior of the asynchronous components so that the data flow specification can be directly mapped into a hardware realization. In addition, we develop a timing model for the basic asynchronous building blocks and show how to derive the timing parameters of a composed systems. This timing model can also be used at the data flow level, allowing designers to explore various design alternatives. We then describe a number of applications of the data flow specification for high-level synthesis such as schemes for resource sharing local transformations for data flow description optimization, and allocation and sequencing of operations for given resources. Finally, we present two examples using this synthesis method. The effectiveness of the data flow specification and performance analysis has been demonstrated from the areas and the simulation of actual layouts generated using an industrial standard cell library and commercial CAD tools.

14. SUBJECT TERMS

Asynchronous circuits/systems, Data flow graph, Token,Micropipelines, Handshaking protocol, Resource sharing, Algorithmic transformations, Sequencing and allocation.

15. NUMBER OF PAGES

73

16. PRICE CODE

17. SECURITY CLASSIFICTION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reoprts. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to stay within the lines to meet optical scanning requirements.

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month,a nd year, if available (e.g. 1 jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element numbers(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract	PR - Project
G - Grant	TA - Task
PE - Program Element	WU - Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the repor.

Block 9. Sponsoring/Monitoring Agency Names(s) and Address(es). Self-explanatory

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of ...; To be published in... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE - See authorities.
NASA - See Handbook NHB 2200.2.
NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.
DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA - Leave blank.
NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17.-19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contins classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

Synthesis of Asynchronous Systems from Data Flow Specifications

Tzyh-Yung Wuu

Information Science Institute
Univ. of Southern California
wuu@lepton.isi.edu

Sarma B. K. Vrudhula
(a.k.a. Sarma Sastry)
ECE Dept.
Univ. of Arizona
sarma@kashi.ece.arizona.edu

Abstract

This report presents a method for automatic synthesis of asynchronous digital systems from high-level data flow specifications. We present an *extended* data flow model that accurately reflects the behavior of the asynchronous components so that the data flow specification can be directly mapped into a hardware realization. In addition, we develop a timing model for the basic asynchronous building blocks and show how to derive the timing parameters of a composed system. This timing model can also be used at the data flow level, allowing designers to explore various design alternatives. We then describe a number of applications of the data flow specification for high-level synthesis such as schemes for resource sharing, local transformations for data flow description optimization, and allocation and sequencing of operations for given resources. Finally, we present two examples, a 16-bit multiplier and a 16-point FIR digital filter, where the number of modules have been altered at the data flow level using this synthesis method. The effectiveness of the data flow specification and performance analysis has been demonstrated from the areas and the back-annotated simulation of actual layouts generated using an industrial standard cell library and commercial CAD tools.

1 Introduction

This paper presents a method for the automatic synthesis of asynchronous digital systems. The input is a data flow specification of the system's behavior and the result is a design with sufficient detail to permit fabrication. Existing approaches focus primarily on the synthesis of asynchronous control circuits. Our work emphasizes asynchronous *systems* such as microprocessors or special purpose processors used in image and signal processing applications, where there is an enormous potential for concurrent computations at the function level. Our approach uses a data-driven model to describe the functional behavior of asynchronous systems. In this model, the data flow specification frees the designer from having to identify concurrent activities and their synchronization explicitly, thus allowing complete exploitation of concurrency. More importantly, our data-driven model allows the various system measures such as delay and area to be incorporated in the high-level specification. This enables the designer to rapidly explore many design alternatives at the data flow level, examining the

tradeoff between performance and area. Furthermore, these high-level design decisions can be replaced by design automation algorithms, namely, high-level synthesis [11, 18].

There are three main aspects of a synthesis system: the *specification*, the *realization* and the *methods*. The *specification* deals with developing a suitable representation of the abstract behavior. The *realization* is a representation of the system in terms of a set of interconnected components. The *methods* are a collection of techniques that translate a specification to a realization.

We start by describing the components of the realization. That is, we first describe the basic building blocks of the asynchronous system. The data communication of these building blocks is controlled by the *handshaking protocol*. We then show that the handshaking protocol can be accurately described by the *token* of a data flow model. Next we describe the abstract specification. Here we present the classical data flow graph (DFG) representation as the behavior specification of asynchronous systems in our synthesis method. With the consideration of hardware (register) cost, we derive an extended data flow graph (EDFG) based on the same token-handshaking model. The structure of the EDFG is very similar to the structure of the DFG. However, the semantics of the EDFG are defined to comply with the behavior of the asynchronous circuits without registers. We also show that each function node in a DFG corresponds to a composition of register nodes and a non-registered node in an EDFG. Thus the EDFG provides a bridge between an abstract specification and the implementation. After having discussed the two ends of the synthesis system, we describe the methods for translating an EDFG to a realization. This is done with respect to a given library of components. Then we develop a timing model for the basic building blocks and show how to derive the timing parameters for any composition of the building blocks. This timing model is applied to the data flow representation. After defining the input specification and its timed behavior model, we present several synthesis methods at data flow level to help designers explore different design alternatives. *Sharing schemes* are templates in a DFG to share a resource or resources by the same type of operations. *Local transformations* perform peephole optimization/reduction in a DFG specification. According to sharing schemes and their performance/area effects, *allocation and sequencing algorithms* allocate operations in a DFG to a given set of modules and order the sequence of operations which share a common module. By applying these synthesis methods, designers may obtain various designs with the objective of minimizing delay, throughput, or area.

Our design procedure of asynchronous systems is shown in Figure 1. This paper focuses on the top half of this design procedure, especially the data flow specification, its timed behavior model, and its relation to the realization and the synthesis methods. The details of realizations and synthesis algorithms are deferred to future papers. Our layout implementation relies on the *MOSIS netlist-to-parts service* [35, 36]. This paper is organized as follows. Section 2 reviews related work. Section 3 identifies the basic building blocks of the realization and maps their behavior into a data flow model. Section 4 describes the data flow specification DFG and the extended specification EDFG. Section 5 constructs a timing model for the building blocks and the function node for the DFG/EDFG. By using this timing model, designers are able to analyze the system performance and other system mea-

sures at data flow level. Section 6 presents several synthesis methods using the DFG/EDFG specification, and their effect in terms of performance and area can be easily determined. Section 7 presents two detailed examples to demonstrate our design method. This includes the DFG specification, various design alternatives that are possible with the given DFG, and mapping the various designs to obtain a number of realizations. These design alternatives are implemented with a standard cell library and their performance and area costs are presented. The last section presents a conclusion for this work.

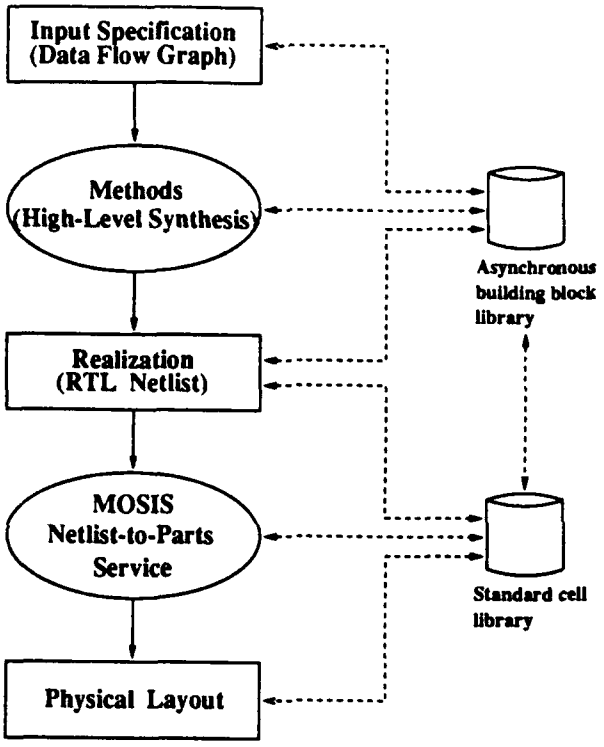


Figure 1: Overview of our design system.

2 Background

Much of the classical work done in asynchronous design has focused primarily on gate level control circuits. Methods for realizing such circuits are based on the Huffman model [10, 17] of a finite state machine (FSM). Such an approach is practical only for relatively small circuits. Moreover, the FSM model cannot describe concurrent behavior at any higher level.

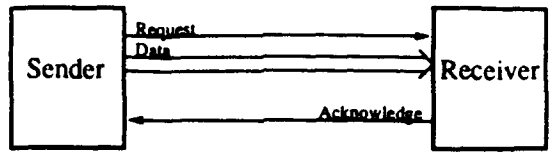
During the past five years there has been a tremendous resurgence of interest in the design of large scale asynchronous system [16] and more recently in the automatic synthesis of such systems [5, 6, 21]. An important aspect of certain types of asynchronous designs is that they

make it feasible to carry out large system design in a *truly* modular fashion by composing independently-designed components and ensuring correctness by construction [26, 30]. In the area of asynchronous design there appear to be two approaches. One approach focuses on the design of reliable asynchronous circuits, e.g., hazard-free asynchronous circuits and delay-insensitive circuits. These methods are based on the manipulation of formal specifications such as signal transition graphs (STG) and Petri nets [7, 14, 15, 21, 23]. The other approach focuses on the synthesis of asynchronous systems by the interconnection of pre-defined asynchronous modules. These methods attempt to translate a high-level language specification such as CSP, CSP-liked descriptions, OCCAM, or Trace structures [1, 5, 6, 9, 30] into a realization. The main task in these synthesis approaches is to correctly decompose/refine the given behavior description into atomic constructs, which have corresponding pre-defined asynchronous modules. However, much work done in the decomposition of asynchronous systems has mainly focused on the synthesis of control circuits. There appears to be little work done in synthesis of both control and data paths of asynchronous systems. In particular, problems related to the incorporation of system level performance measures in the high-level specification and the synthesis of asynchronous systems that take into account constraints on the availability of resources have not been dealt with adequately. It is necessary for the designer to explore these various design alternatives. The goal of our research is to tackle these design issues at system level.

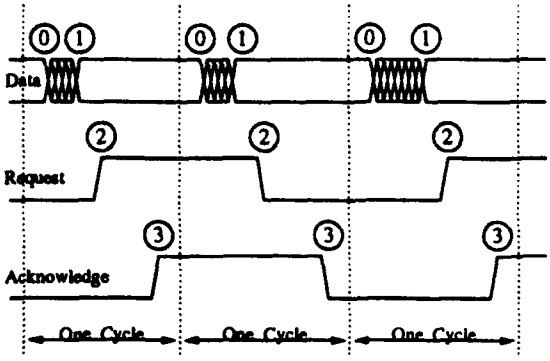
Our approach the resembles those approaches presented in [13, 19, 20] in the design specification and the mapping method. However, our approach is different from theirs in the following sense. Their basic modules are synchronous circuits. In their approach, each node in the data flow graph maps to a unique hardware module. Due to the physical limitation of VLSI, they implement module selection techniques to reduce the area of a design. They partition the nodes of a data flow graph into multiple groups so that each group can be implemented on a chip. Our basic modules are instead asynchronous circuits. In our model, a token in a DFG represents not only data but also the synchronization state between modules, i.e., the state of handshaking signals between modules. Despite the difference of the basic circuit models between their approach and our approach, the ideas of module selection and system partition are important and applicable in our design procedure. However, we emphasize the module utilization more in this research. In order to satisfy constraints on area and/or performance we develop techniques for scheduling and allocation over the nodes in a DFG. Therefore, a node representing an operator (or a hardware module) in the final EDFG may correspond to more than one node representing an operation (or a computation) in the original DFG specification.

3 Hardware Implementation and Data Flow Model

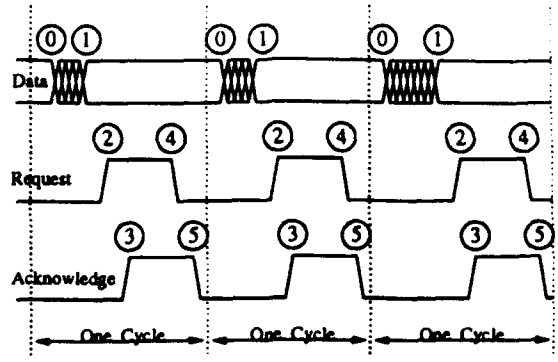
The hardware model that is employed here is based on Sutherland's Micropipelines [28]. This model assumes that request signals are bundled with the data signals to ensure proper operation, namely, the ***bundled data convention.*** Unlike speed-independent and delay-insensitive designs, the micropipeline model requires determination of the delays in the computational blocks. This does not pose any serious problem, as this can be done in a manner similar to



(a) Data transfer between two blocks



(b) Two-phase handshaking



(c) Four-phase handshaking

Figure 2: Data transfer and handshaking protocol.

the conventional design of synchronous systems.

A system consists of a collection of functional *blocks* with data transfers taking place between two or more functional blocks or between a functional block and the surrounding environment. Data transfers between any two blocks rely on a handshaking protocol. Each block will be activated whenever its input data is available. Therefore, the operations of functional blocks in micropipelines are asynchronous, concurrent, and data-driven.

3.1 Data Transfers and Handshaking Protocols

The handshaking protocol used in our design method can be a two-phase and/or a four-phase handshaking protocol. These are shown in Figure 2.

Our current implementation follows the two-phase handshaking protocol with the bundled data convention [26, 28]. Referring to Figure 2(b) we see that there are three events in each cycle of data transfer. First, valid data is put on the data bus by the sender. Second, a signal transition is activated on the request line by the sender to notify the receiver that data is available. Third, a signal transition is activated on the acknowledge line by the receiver to notify the sender that the data has been received so that another cycle of the data transfer can begin.

For the four-phase handshaking protocol shown in Figure 2(c), the request line and the

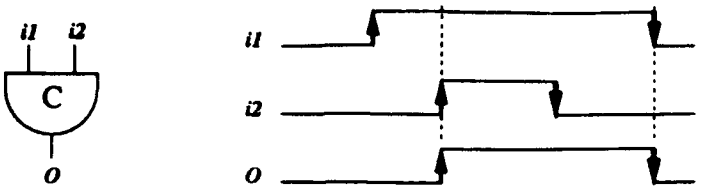


Figure 3: The behavior of Muller C-element.

acknowledge line are initialized to 0 at the start of each cycle of data transfer. The first three events are the same as those in the two-phase handshaking protocol. The next two events are that the request is reset to 0 by the sender and that the acknowledge is reset to 0 by the receiver. In terms of the period of data validation, there two kinds of conventions for the four-phase handshaking protocol [4]. In the *narrow convention*, the sender holds the data valid from the rising request signal to the rising acknowledge signal. In the *broad convention*, the sender holds the data valid from the rising request signal to the falling acknowledge signal. Although our current implementation follows the two-phase handshaking protocol with the bundled data convention, a system may contain different protocols for different data transfers in its implementation, as long as the sender and the receiver of each data transfer follow the same protocol [1].

3.2 Realization of a Basic Block

A functional block as proposed by Sutherland [28] has the structure shown in Figure 5(a). There are three basic elements in this structure. The Muller C-element, represented by a "C" gate, is used to control the handshaking protocol. The asynchronous register, represented by a "reg" block, is used to capture and pass input data. The computational part, represented by a "Logic" block, is used to perform the functional computation for the structure, e.g., addition. The oval node in this structure represents an added delay, which is used to ensure that the output data transfer satisfies the bundled data convention.

Muller C-element There are two inputs and one output for a Muller C-element. The output of a C-element is 1 if all the inputs are 1, and it is 0 if all the inputs are 0; otherwise its value remains unchanged [26]. A two input C-element can be viewed as a logical *and* of two *events*, where an *event* can be a 0-1 or a 1-0 transition [28]. This behavior is shown in Figure 3 [2].

Asynchronous register The asynchronous register proposed by Sutherland is defined as follows. There are four terminals for the register. "Di" and "Do" are the data input terminal and the data output terminal of the register; "c" and "p", which are called capture and pass

[1]The broad convention and the narrow convention of the four-phase handshaking protocol are two different protocols.

[2]Circuit delay is not considered in this figure.

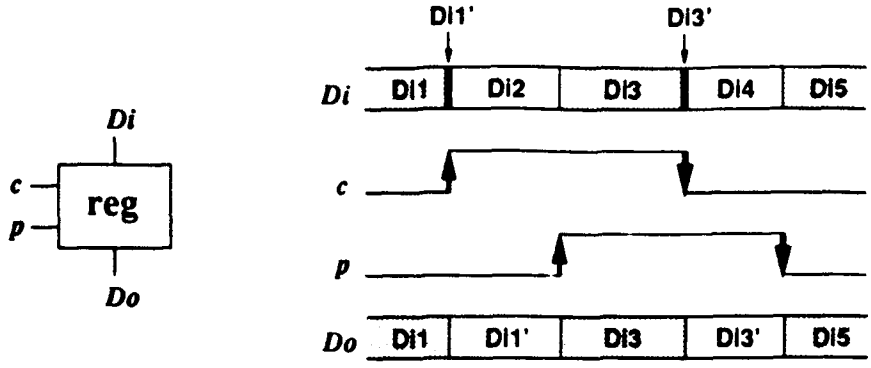


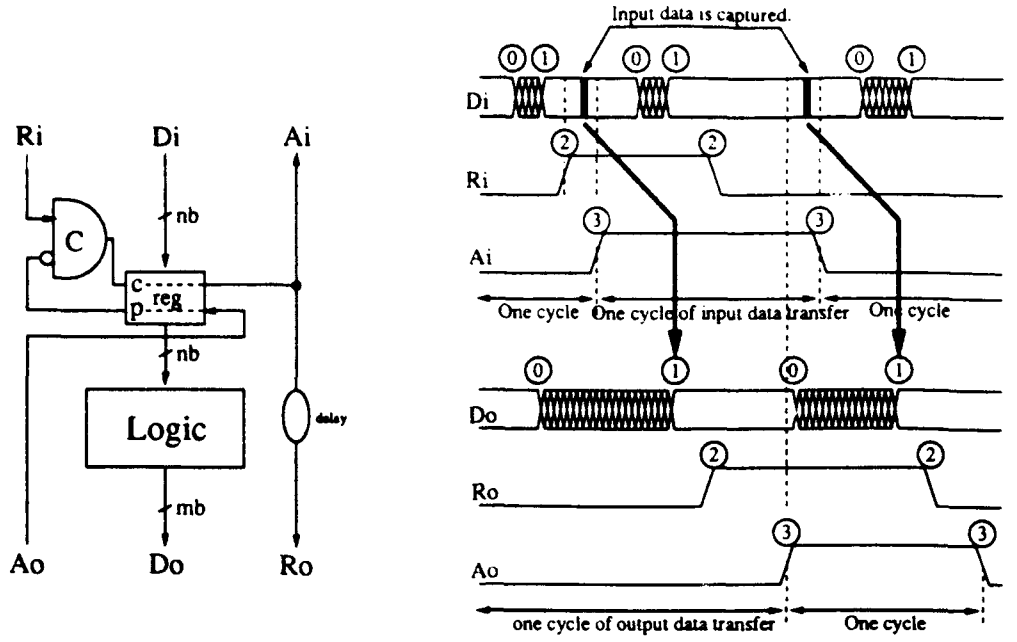
Figure 4: The behavior of asynchronous register.

respectively, are two one-bit control signals of the register. If the value of "c" equals the value of "p", then the value of "Di" is passed to "Do"; otherwise the value of "Do" remains unchanged. Operationally, "c" and "p" are initialized to 0, then signal transitions (events) occur at "c" and "p" in the sequence of cpcp.... This behavior is shown in Figure 4 [3]. In the operation of the asynchronous register, event "c" always captures input data to let the output hold the last input value before event "c", e.g., Di1' and Di3' in Figure 4 are captured by "c" events; event "p" starts the passing mode of the register, e.g., Di1, Di3, and Di5 in Figure 4 are passed to "Do".

Computational part The computational part can be implemented by combinational logic, however, added delay is required to ensure the handshaking protocol. The computational part can also be implemented by *differential cascode voltage switch logic* (DCVSL) without added delay [21]. DCVSL is suitable for four-phase handshaking operation; therefore, we need to have two-to-four and four-to-two phase change circuits to make this kind of circuit useful in two-phase design. The bundled data convention and the bounded delay model are used here primarily to save silicon area and the design time for the computational parts.

Combining Muller C-elements and asynchronous registers forms the pipeline structure of asynchronous systems, i.e., *micropipelines* [28]. We take a single stage from Sutherland's micropipelines as a basic functional block in our system, and it is shown in Figure 5(a). "Ri", "Ai", and "Di" correspond to input request signal, input acknowledge signal, and input data signals; "Ro", "Ao", and "Do" correspond to output request signal, output acknowledge signal, and output data signals. The input/output behavior of the basic block is shown in Figure 5(b). Operationally, "Ri", "Ai", "Ro" and "Ao" are initialized to 0. Notice that there is an inversion at one of the inputs at Muller C-element; it means that there is an initial event at that input. Therefore, event "Ri" will generate an event at the output of the Muller C-element, which will capture the input data "Di". After passing through the register, this

[3]Circuit delay is not considered in this figure.



(a) Block diagram

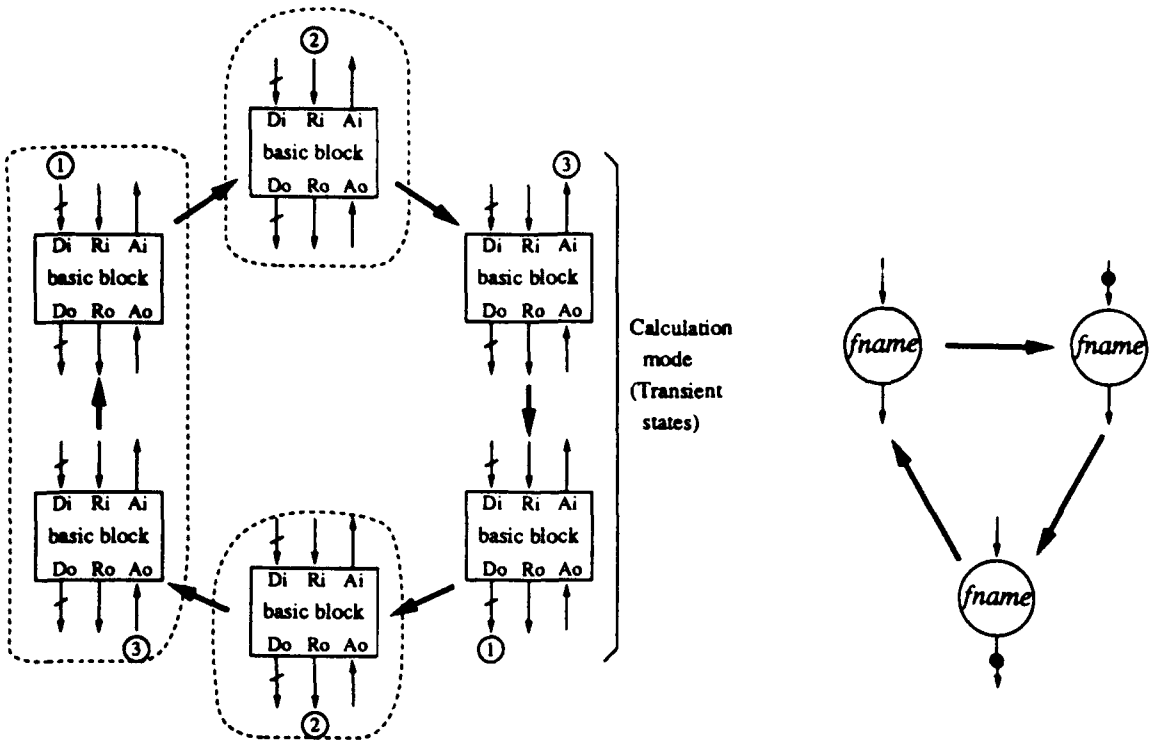
(b) Timing diagram

Figure 5: The basic block of micropipelines.

event will pass to "Ai" to notify the input data transfer that "Di" is stored, i.e., its value can be change. In other words, one cycle of input data transfer is complete. The captured input is presented at the output of the register, which is then operated on the computational part. The added delay is to ensure that the valid data is produced before the arrival of event "Ro". Therefore, the added delay equals the critical path delay of the computational part plus some safe margin delay. Event "Ao" will complete one cycle of output data transfer, and it allows the register to pass new input data to the functional block. After "p" of the register receives an event from "Ao", the Muller C-element receives an (initial) event to allow another cycle of input data capture. In case the new "Di" and "Ri" have arrived before the transfer of output data is completed, the Muller C-element will wait until another input of the C-element receives an (initial) event from "Ao" through "p" of the register.

3.3 Data Flow Model for Basic Blocks

The main reason to use data flow specification in our system is that the behavior of basic blocks of asynchronous systems can be described by a data flow model. We view each basic block as a functional unit. Data is available at the input of the block and is captured by the block, data is produced at the output of the block, which becomes the input data of another functional block. The behavior of the basic block is analogous to the behavior of a functional node in a data flow graph, which *absorbs* input tokens and *generates* output tokens. This



(a) The behavior of a basic block

(b) Data flow model

Figure 6: Data flow model for basic blocks.

analogy is shown in Figure 6.

Figure 6(a) shows a sequence of states to describe the same input/output behavior of the basic asynchronous block that Figure 5(b) describes, where labels 1, 2, and 3 represent the sequence of events – data available, the request signal transition, and the acknowledge signal transition of the two-phase handshaking protocol. Each state in Figure 6(a) represents that the block has just received or produced an event, which is denoted by a numbered circle, and is waiting for next event. The two states which are grouped together at the left of Figure 6(a) correspond to an idle functional node in the data flow model. Although there is input data available at the last state of this group, the block has not been notified of the availability of input data by the external world [4]. Only when event "Ri" is activated by the external world, the basic block knows that there is data available at "Di". Therefore, the top center state in Figure 6(a) corresponds to a data flow function node with an input token. After the block captures the input data and completes its calculation, the block produces an output data, and the external world is notified by event "Ro". Therefore, the state at bottom center of Figure 6(a) correspond to the output token generated in the data flow model. After the external world releases the output data by activating event "Ao", the block becomes idle

[4]The external world means the surrounding environment of the basic block.

again. Two *transient states* in Figure 6(a) are not mapped into the data flow model, and they represent the functional operation in the real circuit which takes time. However, they can be ignored in the high-level data flow model and are represented by a proper timing model for system analysis.

Handshaking protocol and token model The key of this analogy is to model the data transfer, which is based on two-phase handshaking protocol, by the token movement in the data flow model. Referring to Figures 2(a) and 2(b), the data transfer between event 2 (Request) and event 3 (Acknowledge) is the state that the data is available on the data bus and is waiting to be captured by the receiver. This state is mapped into an appearance of a token between two function nodes in the corresponding data flow graph; the token is generated by the function node which corresponds to the sender block, and it will be absorbed by the function node which corresponds to the receiver block. Later, we will derive the extended data flow graph based on the same token model.

4 Data Flow Specification

The data flow graph (DFG) is used as the input specification, and it is based on the token model used in data flow computing [8]. In this section, we briefly describe the structure and semantics of the DFG used in our synthesis system.

A DFG is a directed graph with typed nodes and port-specific arcs, where *port* refers to the input/output terminals of a node. Each node in a DFG belongs to a finite set of node types which represent the *basic constructs* of the DFG specification. Each directed arc in a DFG connects a specific output port of a node to a specific input port of a node. The semantics of a DFG are expressed by the movement of *tokens*. A token represents the presence of data on the corresponding input. A node is *activated* when all its necessary input arcs have tokens. An activated node *computes* or *fires* by absorbing all the tokens on its inputs and placing tokens on its outputs. There is no notion of synchronization among activated nodes, as these nodes operate asynchronously and concurrently [8].

4.1 Basic Constructs

By considering area/performance efficiency of asynchronous block implementation, we have generalized and enriched the basic DFG constructs, which are shown in Figure 7, from the conventional data flow specification. For example, the conventional data flow specification often uses binary input (or output) control constructs such as the Distributor and the Selector. To distribute one data to one of eight destinations, we would need to use three levels of these two-input Distributors. In terms of delay and area consumption, we found it is more efficient to implement a single block to handle one-to-eight distribution than to use three levels of the two-input Distributors. To distinguish from the conventional constructs in data flow specification, we prefix the names of these multiple input/output control constructs with "M". These enhancements imply that the set of basic constructs may grow in the future as long as the new constructs satisfy the data flow model and they are needed in the description

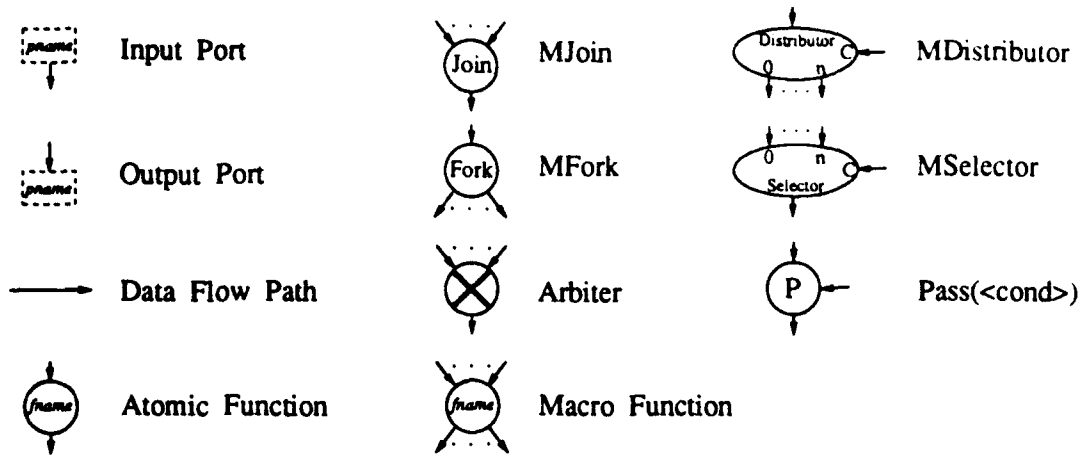


Figure 7: Basic constructs of the DFG.

of asynchronous systems [5]. The behavior of the basic constructs shown in Figure 7 are given below.

- **MJoin:** If there is a token at each input, MJoin absorbs all the input tokens and generates an output token. The output token represents all data values from all input tokens.
- **MFork:** If there is an input token, MFork absorbs the input token and generates a token on each of its output with the same data value as on the input.
- **MDistributor:** If there is a token at the data input port and a token at the condition input port carrying the value m, MDistributor absorbs both input tokens and generates a token at output port m with the same value as on the data input port.
- **MSelector:** If there is a token at input port m and a token at the condition input port carrying the value m, MSelector absorbs both input tokens and generates an output token with the same data value as on input port m.
- **Pass(⟨cond⟩):** If there is a token at the data input port and a token at the condition input port, Pass(⟨cond⟩) absorbs both input tokens, and generates an output token with the same data value as on the data input if the condition data equals ⟨cond⟩.
- **Arbiter:** If there exist token(s) at the input port(s) and there is no token at the output port, one and only one input token is absorbed and passed to the output port.
- **Atomic functions:** These represent computational nodes, e.g., adders, multipliers, and so on.

[5] The minimum set of basic constructs is not very meaningful for a hardware description language.

- **Macro function:** A macro function represents a function defined by another data flow graph, and it supports hierarchical description.

There are three rules regarding the data flow specification and its behavior model:

1. At most one token is allowed on an arc at any time.
2. Every basic construct can absorb tokens from its input port(s) only if no token is present at any of its output arcs. In other words, no tokens are allowed to accumulate in any of the basic constructs.
3. No recursive (macro) function is allowed in our system.

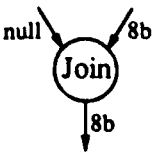
4.2 Data Types

Since the goal is to transform or translate DFG descriptions into hardware realizations, each data item has a fixed format as specified by the *data type*. For example, the input and the output of a 16-bit adder have certain data formats, e.g., the input contains two 16-bit data, and the output contains a 1-bit carry-out and a 16-bit sum. There are three basic data types:

1. A null data type is denoted by **null**.
2. A set of n-bit wire data types is denoted by n**b**, where n is a positive integer.
3. Group data types are denoted by $\langle g_1, g_2, \ldots, g_m \rangle$, where each g_i is a null data type, or a wire data type, or another group data type.

The following items in a DFG are data typed: input/output ports, directed arcs, and tokens. If an output port is connected to an input port through a directed arc, the output port, the input port, the directed arc, and tokens which flow through the arc should have the same data type. The data type of an input/output port depends on what kind of function this node is. For example, a 16-bit adder may have input data type ⟨16b,16b⟩ and output data type ⟨1b,16b⟩.

Except atomic functions, macro functions, and MJoin, input ports, excluding the conditional input, and output ports of any construct have the same data type. MJoin absorbs tokens from all inputs and generates a token representing all input tokens. If n inputs of MJoin from left to right have data types $t_1, t_2, \ldots, t_n$, then the data type of the output is $\langle t_1, t_2, \ldots, t_n \rangle$. During the manipulation of a DFG, a data type may be reduced to an equivalent data type. $\langle g_1 \rangle$ is equivalent to g_1, and $\langle g_1, g_2, \ldots, g_{j-1}, g_j, g_{j+1}, \ldots, g_m \rangle$ is equivalent to $\langle g_1, g_2, \ldots, g_{j-1}, g_{j+1}, \ldots, g_m \rangle$ if g_j is a null data type. A data type is *primitive* if it is neither a group data type containing null data type(s) nor a group data type containing only one data type. Every data type is equivalent to a unique primitive data type. Hence only primitive data types are considered during the manipulation of DFG constructs, e.g., MJoin shown in Figure 8.



Join of null and 8b is <null,8b>
<null,8b> is equivalent to <8b>.
<8b> is equivalent to 8b.

Figure 8: Example: manipulation of data types.

There is no distinction between a data signal and a control signal in our system. A control signal in the conventional data flow graph is an one-bit data signal which carries either *true* (logic 1) or *false* (logic 0), i.e., its data type is 1**b**. Furthermore, we generalize control signals to be n**b**, which can control multiple (2^n) input/output choices.

In a DFG, each token carries a data value. The data value of a type **null** token is $\langle\rangle$. A valid value of a type n**b** token is $\langle x_1, \ldots, x_n \rangle$, where each x_i for $1 \leq i \leq n$ is any valid value on a wire, e.g., 0, 1, and X. A valid data value of a group data type $\langle g_1, g_2, \ldots, g_m \rangle$ token is $\langle v_1, v_2, \ldots, v_m \rangle$, where v_i is a valid value of data type g_i for $1 \leq i \leq m$. For example, the valid values of a 8b token are $\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8 \rangle$'s with each $x_i \in \{0, 1, X\}$ for $1 \leq i \leq 8$. Similarly, the valid values in ⟨4b,4b⟩ are $\langle \langle x_1, x_2, x_3, x_4 \rangle, \langle y_1, y_2, y_3, y_4 \rangle \rangle$'s with $x_i, y_i \in \{0, 1, X\}$ for $1 \leq i \leq 4$. Both of the above two data types are implemented as eight wires in hardware, but they may represent different meanings in the DFG. For example, the former may be an 8-bit integer, and the latter may be two 4-bit integers.

4.3 The Need of Extended Data Flow Graph

In Section 3.3 we use a data flow graph to model basic blocks in micropipelines. Conversely, we can translate a DFG description into an asynchronous system, which is composed of basic blocks. However, each input port of every block needs a register to latch data, so this kind of implementation may result in many registers. For example, the two-input addition DFG description in Figure 9 can be directly translated into the implementation by mapping nodes MJoin and ADD into blocks. In this example, there are two levels of registers. In terms of area and performance efficiency, we don't need both levels of registers. Therefore, removing the input register of the ADD block yields an implementation with better performance and smaller area.

4.3.1 Register Blocks and Computational Blocks

In order to reduce the cost of registers, we separate registers from basic blocks of micropipelines. Two basic blocks are defined: the register block and the computational block, shown in Figure 10(a). Their input/output behaviors are shown in Figure 11, where the variables D_{sfl}, D_{sbl}, D_{sp}, D_{fp}, and D_{bp} are delays between events which will be defined later.

In terms of input/output events, the behavior of the register block is exa ly the same

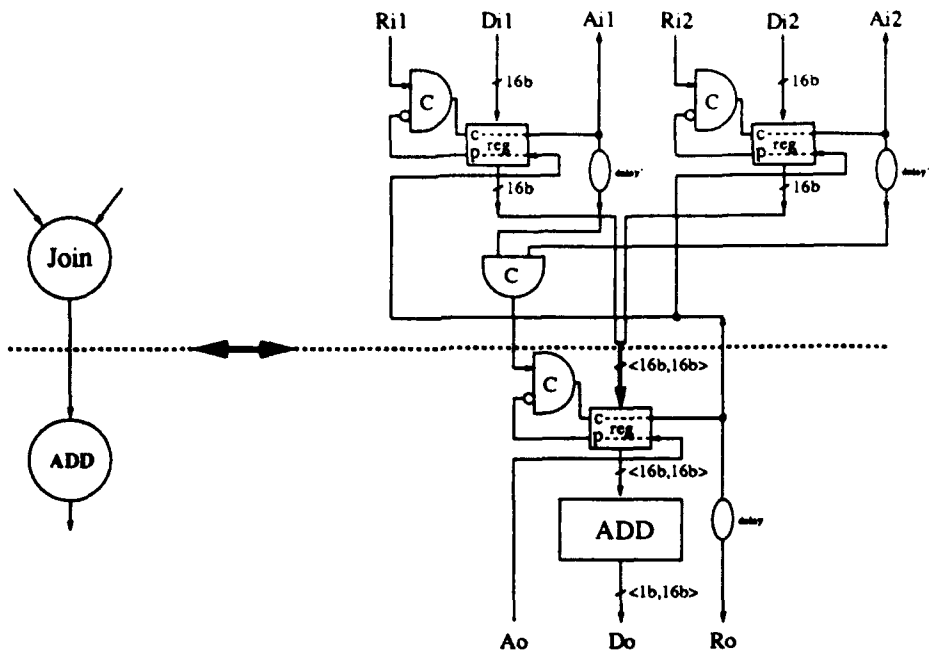


Figure 9: Two-input addition.

as the basic block of micropipelines. On the other hand, the input events and the output events of a computational block are closely related. Because there is no storage in the computational block, the input data cannot be released until the output data is released. Therefore, a complete output event cycle of the two-phase handshaking protocol occurs between event "Ri" and event "Ai" of the input event cycle. In other words, event "Di" is followed by event "Ri", which is followed by event "Do", which is followed by event "Ro", which is followed by event "Ao", which is followed by event "Ai". Figure 12(a) shows the sequence of input/output events.

4.3.2 Extended Data Flow Model for Computational Blocks

Since the behavior of the functional block without storage differs from the original functional block of micropipelines, a *phantom* node is used to represent the corresponding data flow construct for the computational block in Figure 10(b). Using the same handshaking protocol-token analogy in Section 3.3, we define the extended data flow model for computational blocks as shown in Figure 12. Each state in Figure 12(a) represents that the block has just received or produced an event and is waiting for the next event. The left two states in Figure 12(a) correspond to an idle phantom functional node in the extended data flow model. When event "Ri" occurs, the computational block is notified by the external world that an input data is available at "Di". Because there is no memory in the computational block, the value of "Di" has to be kept valid by the external world until the corresponding output data is released, i.e., event "Ao" occurs. Therefore, the right three states in Figure 12(a) correspond to a phantom functional node with an input token. When valid data is produced at "Do", the

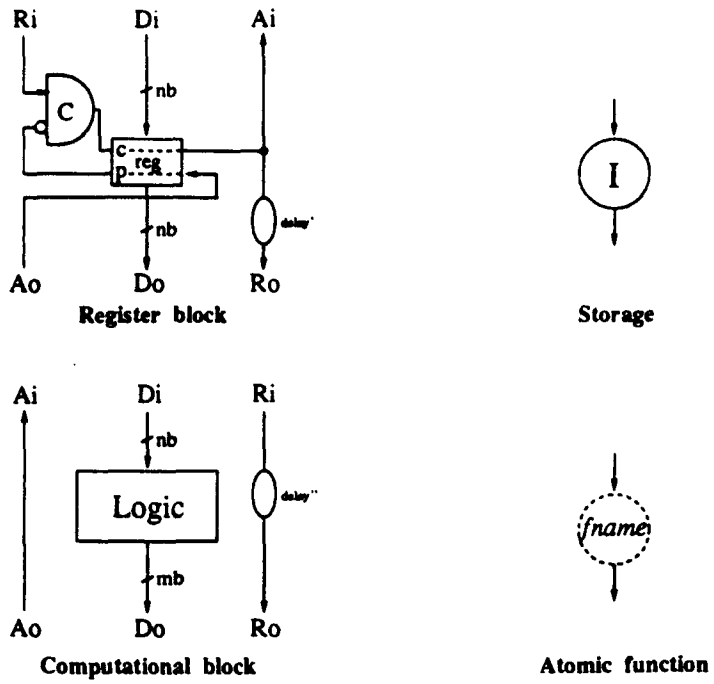


Figure 10: The structure of the basic blocks.

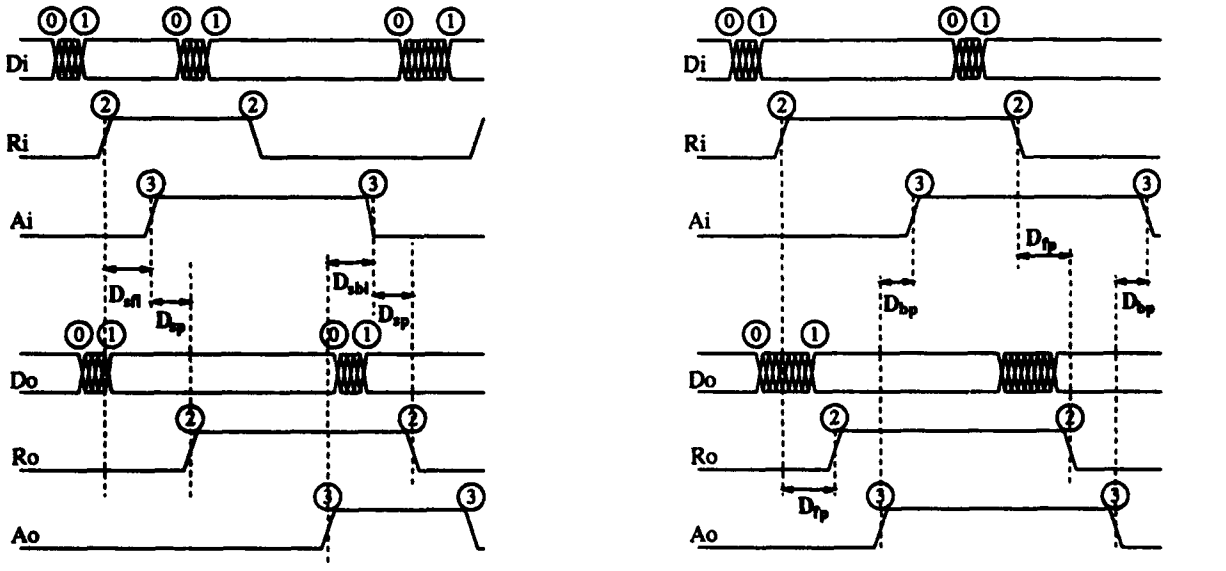
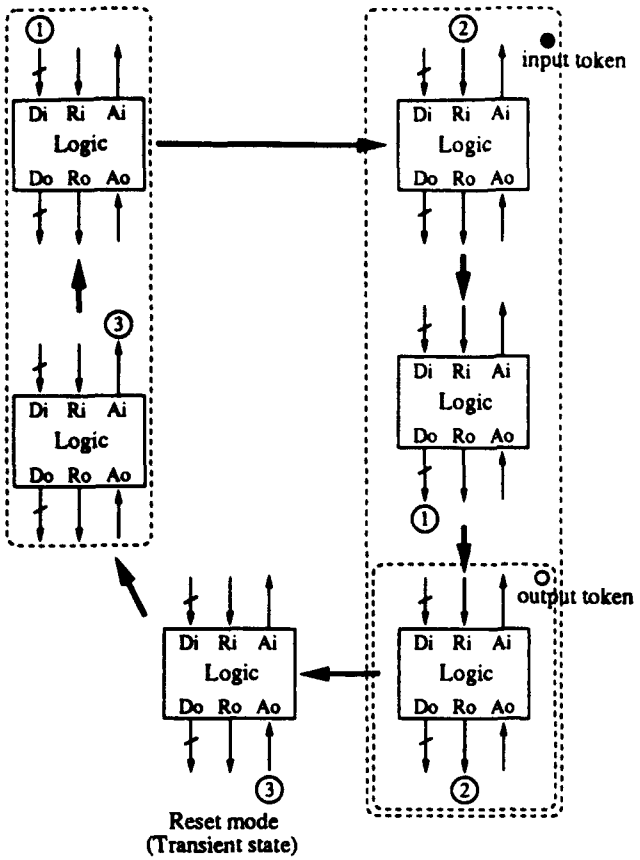
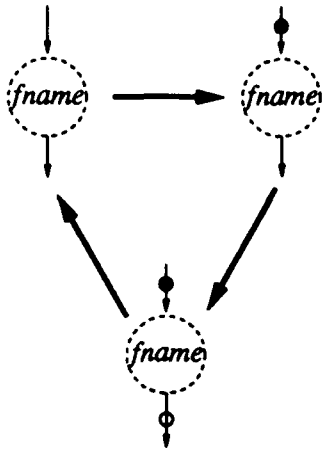


Figure 11: The behavior of the basic blocks.



(a) The behavior of a computational block



(b) Extended data flow model

Figure 12: Extended data flow model for computational blocks.

external world is notified by event "Ro". Therefore, the bottom right state in Figure 12(a) corresponds to a phantom functional node with an output token. After the external world releases the output data by activating event "Ao", the computation block activates event "Ai", and the block becomes idle again. Figure 12(b) is the corresponding extended data flow model for the computational block. Unlike the conventional data flow model, an input token is not absorbed when the corresponding output token is produced; the input token is removed when the corresponding output token is absorbed by the external world. The output token looks like an extension of the input token through the phantom node, so the output token is called an *extended token* (of the input token). One transient state in Figure 12(a) is not mapped into the extended data flow model, and the computational block is reset in this state. In the implementation shown in Figure 10(a), "Ai" is directly connected to "Ao", so this transient state takes zero delay[6]. However, it may remain some time in this reset state if the computational block is implemented by DCVSL. As stated previously, the transient state can be ignored in the high-level data flow model and is taken care of with a

[6]Assume that wiring delay can be ignored.

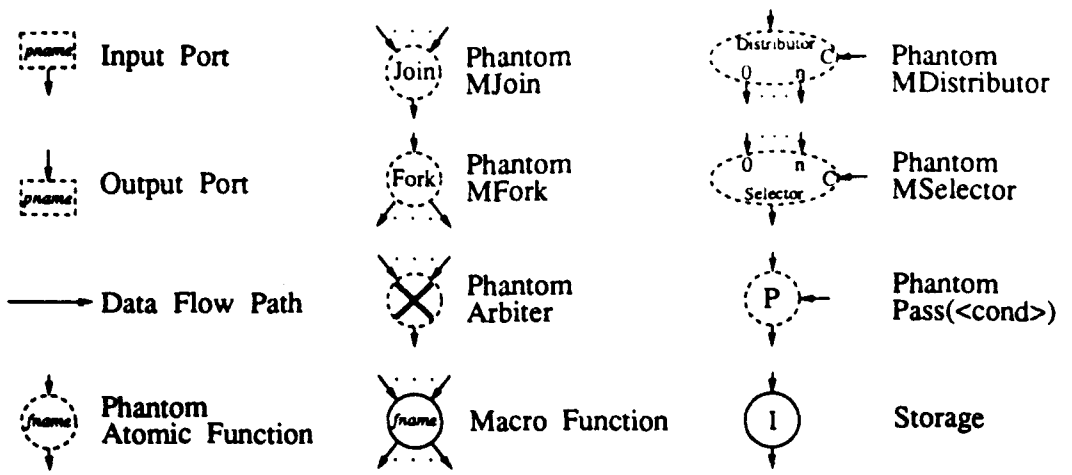


Figure 13: Basic constructs of EDFG.

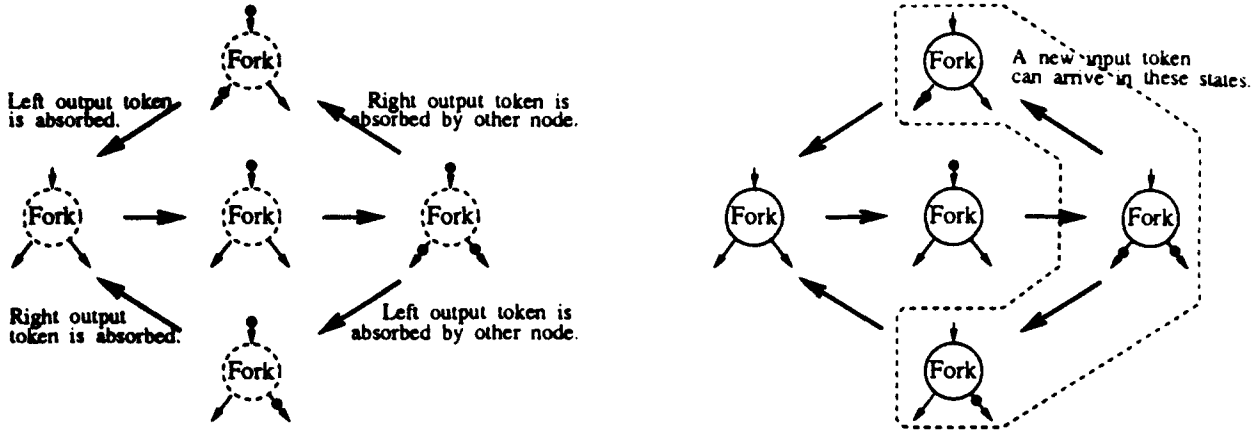
proper timing model for system analysis.

4.4 Extended Data Flow Graph

After separating the register from the micropipeline structure, we now describe a simple but very useful extension to the DFG to describe these new basic blocks. The result is called an *extended data flow graph* (EDFG), which provides a bridge between the abstract DFG specification and the circuit implementation. The set of basic constructs in EDFG are shown in Figure 13.

Syntactically an EDFG is the same as a DFG. However, the semantics of an EDFG are defined so as to comply with the hardware behavior of asynchronous circuits. The essential difference is in the rules that govern the movement of tokens. Based on the extended data flow model described in Section 4.3.2, the behavior of EDFG is described as follows. There are two kinds of tokens, namely, regular tokens and extended tokens. Both kinds of tokens represent where data is available. A regular token represents the data that is the direct output token of a register, and it is denoted by a dark circle. An extended token represents the data that is the output token of a non-register node, and it is denoted by a circle. The behavior of Storage, the only non-phantom construct in the EDFG, is the same as the behavior model described in the DFG; a Storage absorbs input tokens, which are either regular or extended, and generates regular output tokens with the same values as the input tokens. The behavior of a phantom construct in the EDFG is similar to the behavior of the corresponding non-phantom construct in the DFG except for the following differences:

- Input tokens to a phantom node can be regular or extended.
- A phantom node generates only extended tokens.
- When a phantom node generates output tokens, it does not absorb its input tokens.



State diagram of phantom MFork

Partial state diagram of MFork

Figure 14: An example – comparison of EDFG to DFG.

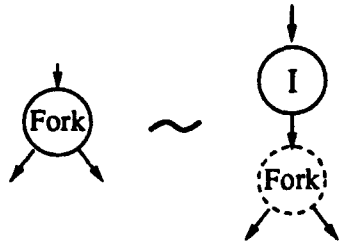


Figure 15: A MFork equals a phantom MFork with an input Storage.

- A token on the input of a phantom node can be absorbed only if all its extended tokens are absorbed.

Figure 14 shows the basic differences in the movement of tokens in an EDFG and the corresponding DFG. After the output tokens are generated, the input arc of an MFork in the DFG can receive a new data token, but the input arc of phantom MFork in the EDFG cannot receive a new data token until all its output tokens are absorbed.

Extended tokens and regular tokens In terms of the availability of data and the analogous meaning to the handshaking protocol, there is no difference between the extended token and the regular token. The purpose of defining extended tokens is to emphasize the semantic difference between phantom nodes and non-phantom nodes, which are used in the conventional data flow graph [8], and it is also to emphasize the relation among input data and output data of phantom nodes.

DFG vs. EDFG A DFG node is equivalent to its phantom counterpart with an input storage at each input port, e.g., the MFork/phantom MFork in Figure 15. Since a DFG description can be replaced by an EDFG description, why do we need the DFG? The first reason is that the DFG lets designers focus on the functional description without worrying about the hardware implementation, e.g., how many adders, where to assign registers, and so on. The DFG is also a well-known language/concept for data flow computing, so it can be easily adopted by designers. Another reason is for the convenience of system synthesis. DFG and EDFG are used in the different stages of asynchronous system synthesis in our system. The DFG is mainly used in the early steps of system synthesis such as sequencing and allocation, mapping of sharing schemes, and local transformation, where a register is assumed for every data transfer. The EDFG is mainly used in the synthesis steps such as register minimization, deadlock prevention, and local transformation before the specifications are mapped to hardware modules. Notice that partitioning the synthesis procedure into steps is not unique and that the tasks of the synthesis steps are usually closely related [11, 18].

4.5 Hardware Translation – Syntax-Directed Method

We adopt the syntax-directed method [5, 6] to realize the physical design from the *extended data flow graph* (EDFG) specification. In this method, each basic construct in the high-level specification is directly translated into a corresponding hardware module. Therefore, the data flow graph not only describes the behavior of a system but also represents the structure of the system. By using this method, the correctness of a hardware implementation is proven by construction. Therefore, the design method mainly focuses on mapping the constructs and the behavior models of the EDFG description into the functional/control blocks of the micropipeline structure, and on reflecting the hardware characteristics of the functional/control blocks to the parameters of the DFG constructs.

Translating EDFG constructs into asynchronous components A path in an EDFG is mapped to a two-phase handshaking data transfer bus, including a data bus (wires) with the same data type, a request line, and an acknowledge line. A token, either regular or extended, on a data flow path corresponds to the state in which the data and the request have been sent but the acknowledge has not been received by the sender. This correspondence allows us to design a component corresponding to each construct in a EDFG, ensuring that the interface requirement is satisfied. Thus there is a one-to-one correspondence between the elements of an EDGF and the hardware modules (see Figure 16). We have developed all the mappings in our cell library [32], e.g., Figure 17 shows a design for the 4-output MFork.

Hardware properties in EDFG Since each basic construct of EDFG is directly mapped into an asynchronous module, the hardware properties of this asynchronous module are attached to its corresponding node in a EDFG description: (1) $D_{sfl}(n_i)$ is the *forward latch time* of register node n_i; (2) $D_{sbl}(n_i)$ is the *backward latch time* of register node n_i; (3) $D_{sp}(n_i)$ is the *propagation delay time* of register node n_i; (4) $D_{fp}(n_i)$ is the *forward propagation delay time* of phantom node n_i; (5) $D_{bp}(n_i)$ is the *backward propagation delay time* of phantom node n_i; (6) $S(n_i)$ is the *area cost* of node n_i; and (6) others. These properties can be used for

system analysis in a high-level data flow description. Furthermore, designers and synthesis algorithms can make design decisions in a high-level data flow description based on these attached hardware properties.

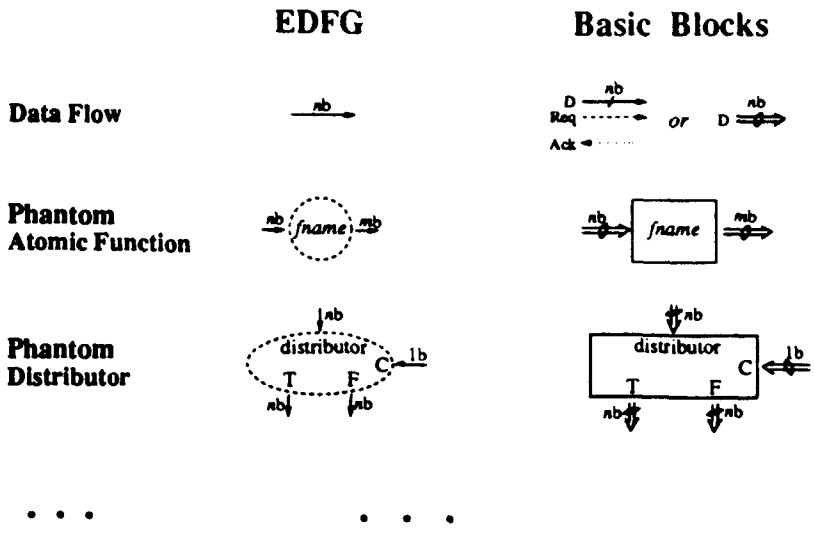


Figure 16: Generic basic blocks.

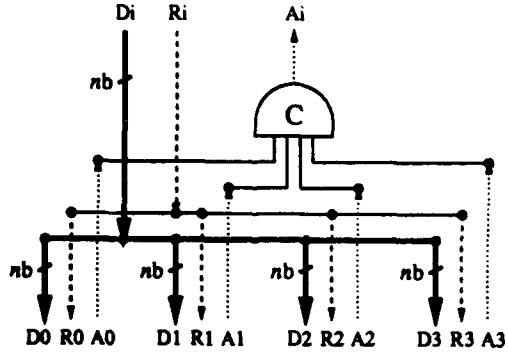


Figure 17: Block design for 4-output MFork.

5 Timing Model for Data Flow Specification

A high-level specification is useful not only to describe the functional behavior of systems, but also to analyze/predict the resulting implementation. In this section we first use timed Petri nets [25] to model the timing behavior of basic blocks. Then we showed that the composition of these timed Petri net models can be used to express the timing behavior

of asynchronous systems which are composed of basic blocks. Based on the timing models derived from timed Petri nets, the timing parameters and the timing behaviors of both DFG and EDFG are defined.

5.1 Timing Behavior Model for Basic Blocks

There are three kinds of basic blocks, referred to as the register block, the computational block and the control block. The circuit structures of register blocks and computational blocks have been shown earlier. Control blocks are those basic blocks corresponding to phantom constructs in EDFG which have more than one input and/or more than one output, e.g., asynchronous blocks of MSelector. If we consider the joining of multiple events in control blocks as "an event", the behavior of a control block is the same as the behavior of a computational block. For example, a two-input MJoin can be activated only if both input request events arrive. The event of "both input request events occur" is equivalent to the input request event of a computational block. Therefore, we only need to model the register block and the computational (non-register) block.

Since the data transfer between blocks follows the two-phase handshaking protocol, the data value is always valid from event request to event acknowledge. Therefore, we only need to model the event of control signals. There are four events associated with the register and computational block:

Ri: *input data ready* – this corresponds to the input "request" signal transition.

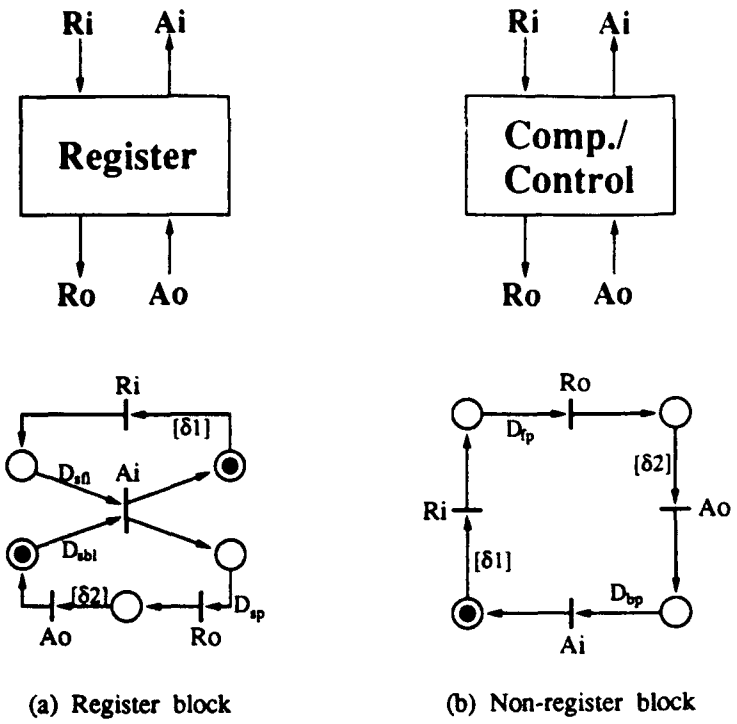
Ai: *input data done* – this corresponds to the input "acknowledge" signal transition.

Ro: *output data ready* – this corresponds to the output "request" signal transition.

Ao: *output data done* – this corresponds to the output "acknowledge" signal transition.

The timing behavior of basic blocks is most easily described using timed Petri nets [25], where transitions represent *input/output (control) events* of the block, and places represent the *conditions* of events in the block. The delay from a place (state) to a transition (event), which is labeled on the arc between them, represents the minimum time interval from when the condition is satisfied to when the transition is activated. The state of the block is represented by the distribution of tokens in the timed Petri net. Figure 18 shows the timed Petri net models for the register block and the computational block, where the tokens of each model represent the initial state of the corresponding block. By simulating the token movement in each Petri net, we can easily find the relation among Figures 18(a) and 6(a) and 11(a) for the register block, and the relation among Figures 18(b) and 12(a) and 11(b) for the computational block.

Timing parameters There is a delay associated with each pair of sequential events. These delays are shown in Figures 11 and 18. The timing parameters for a register block are defined as follows:



(a) Register block (b) Non-register block

Figure 18: Timing model for asynchronous blocks.

D_{sfl} : *forward latch time* is the time for the register to latch the input data when the register is ready and the new data just arrives. This corresponds to the delay from Ri to Ai in the Petri net, where the post-Ri condition represents that input data is ready.

D_{sbl} : *backward latch time* is the time for the register to latch the input data when the input data is ready and the register just becomes available. This corresponds to the delay from Ao to Ai in the Petri net, where the post-Ao condition represents that the register is ready.

D_{sp} : *propagation delay time* is the time from when the input data is latched to when the output data becomes valid. This corresponds to the delay from Ai to Ro in the Petri net.

The timing parameters for a non-register block are defined as follows:

D_{fp} : *forward propagation delay* is the time from when all required input data are valid to when all corresponding output data are valid. This corresponds to the delay from Ri to Ro in the Petri net.

D_{bp} : *backward propagation delay* is the time from when the output data is being acknowledged to when the input data being acknowledged. This corresponds to the delay from Ao to Ai in the Petri net.

In addition to the basic delay parameters associated with each block, there are two delays associated with the environment. These are denoted by $\delta 1$ and $\delta 2$. $\delta 1$ is define as the delay from Ai to Ri. This is the time between the input acknowledge event (completion) to the next input request event (starting). Similarly, $\delta 2$ is defined as the delay between Ro to Ao. This is the time from the output request (starting) to the next output acknowledge (completion). Both $\delta 1$ and $\delta 2$ depend on the response from the environment of the block and are not delays constrained by the hardware implementation of a block, so we bracket the notations.

5.2 Timing Behavior of Composed Blocks

Two kinds of behavior models, the register model and the non-register model, have been defined to describe the behavior of basic asynchronous blocks. If we can derive the composed behavior of these two models in timed Petri nets, we will be able to derive the behavior for any given system. There are four possible combinations of these two models or these two kinds of basic blocks:

1. The output of a register block is connected to the input of a non-register block.
2. The output of a register block is connected to the input of a register block.
3. The output of a non-register block is connected to the input of a non-register block.
4. The output of a non-register block is connected to the input of a register block.

Let B1 and B2 be two basic blocks. Let event E of block B denoted by E(B), e.g., Ri(B1); let timing parameter D_{xx} of block B denoted by $D_{xx}(B)$, where xx is either of *sfl*, *sbl*, *sp*, *fp*, or *bp*, e.g., D_{sfl}(B1). When B1 and B2 are connected with the outputs of B1 feeding the inputs of B2, then synchronization between these two blocks means that Ro(B1) is Ri(B2) and Ao(B1) is Ai(B2). The behavior of a composed block can be generated by merging the timed Petri nets of B1 and B2 with Ro(B1) $\equiv$ Ri(B2) and Ao(B1) $\equiv$ Ai(B2). The timed Petri nets of the four possible combinations are are shown in Figure 19. A formal proof of the correctness of this composition is not presented in this report. These compositions will be used to model the delay parameters and the performance analysis in the data flow specification of the following discussion.

5.3 Performance analysis of linear pipelines

Two measures are defined to evaluate the performance of a system, the *completion time* and the *throughput rate*. The completion time is a measure of how long it takes to complete the execution of a set of data from inputs to outputs of the system. The throughput rate is a measure of how many sets of data can be processed by the system per time unit in steady state. The inverse of the throughput rate is called the *pipeline period*. Let the completion time, pipeline period, and throughput rate be denoted by L, P, and R respectively. By formulating the performance measures for linear pipelines at the block level, we can find a proper timing model for the high-level data flow specification.

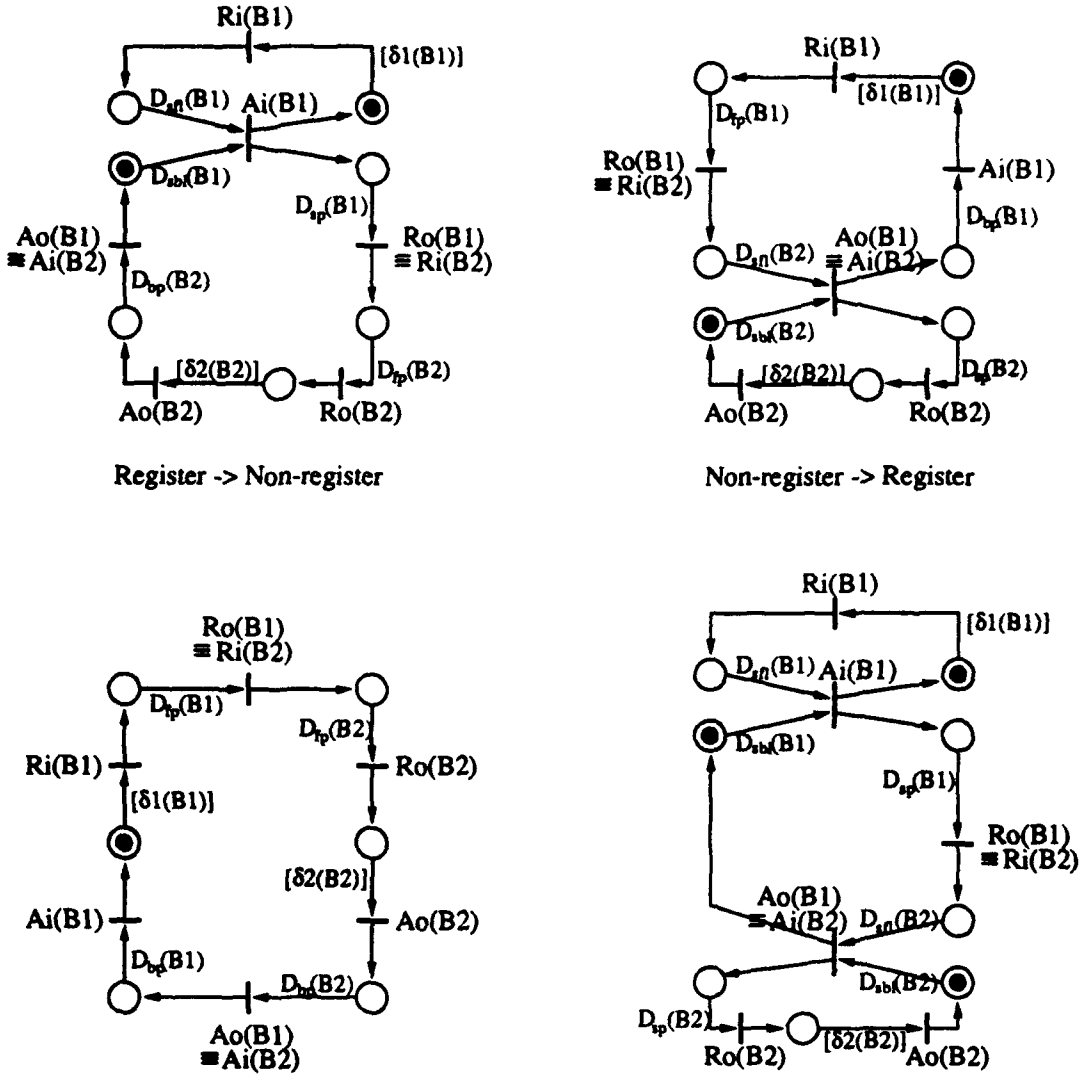


Figure 19: The behavior of composed blocks.

A linear pipeline is a series of computations divided by registers. Although many systems are not linear pipelines, each input-output computation path in a system can be viewed as a linear pipeline. Unlike synchronous systems, the computation time between two consecutive registers is not fixed. We will analyze the performance for a *stage*, and then expand the analysis to the performance of a pipeline.

5.3.1 Performance analysis for a stage

By observing the timed Petri net descriptions in Figure 18 and the descriptions for the composed block in Figure 19, we found that the event Ai of a register breaks the input and the output of the register into two loops, with Ai being the join event of the two loops. We define the parts of hardware described by a loop of events in a Petri net as a *stage*. Two timing parameters are defined for each stage. They are the *forward propagation delay time* and the *backward propagation delay time*, which are denoted by FP_i and BP_i respectively for stage i. The forward propagation delay time of a stage is determined by the timing delay from Ai of the stage's input register to Ai of the stage's output register. The backward propagation delay time of a stage is determined by the timing delay from Ai of the stage's output register to Ai of the stage's input register. Figure 20 (a) is a simple asynchronous system with computational blocks Comp1 and Comp2 between registers Reg1 and Reg2. Figure 20 (b) is the composed behavior of this system. In this system, the output of Reg1, and Comp1, Comp2, and the input of Reg2 form a stage, and the input of Reg1 and the output of Reg2 also form a stage. The three stages from input to output are labeled stages 0, 1, 2, whose forward and backward propagation delay times are formulated as follows. (Note that the input $\delta 1$ (of Reg1) and the output $\delta 2$ (of Reg2) are always assumed to be zero when we measure the performance of a system. In other words, new data is fed into the system as soon as the input register is free; output data is removed as soon as it is available.)

$$
\begin{aligned}
FP_0 &= D_{sfl}(Reg1) \\
BP_0 &= 0 \\
FP_1 &= D_{sp}(Reg1) + D_{fp}(Comp1) + D_{fp}(Comp2) + D_{sfl}(Reg2) \\
BP_1 &= D_{bp}(Comp2) + D_{bp}(Comp1) + D_{sbl}(Reg1) \\
FP_2 &= D_{sp}(Reg2) \\
BP_2 &= D_{sbl}(Reg2)
\end{aligned}
$$

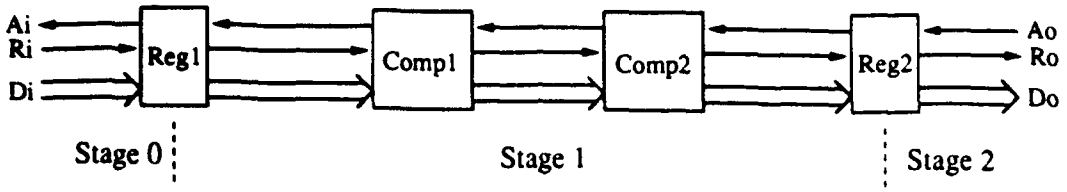
By using these parameters, the timing diagram of this system is obtained by simulation, and it is shown in Figure 21. The measures of L, P, and R can be formulated as follows.

$$L = FP_0 + FP_1 + \max\{(FP_2 + BP_2), BP_1\} \quad (1)$$

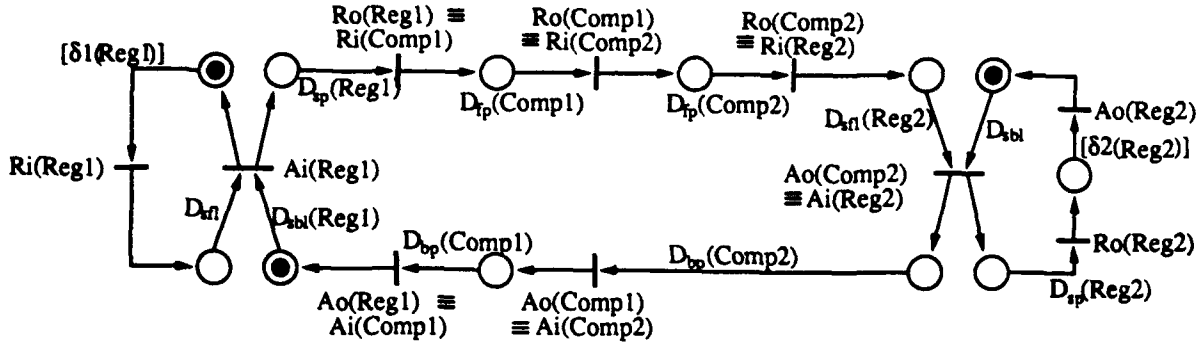
$$P = FP_1 + BP_1 \quad (2)$$

$$R = 1/P \quad (3)$$

There is one further observation from the Petri net description of Figure 20. There is always one and only one token in each loop of the Petri net. The minimum time for the token moving around the loop in any stage i is $(FP_i + BP_i)$, so the lower bound of pipeline period is $(FP_i + BP_i)$. In other words, the throughput rate of stage i is less than or equal



(a) Block diagram



(b) Timed Petri net description

Figure 20: A simple asynchronous system.

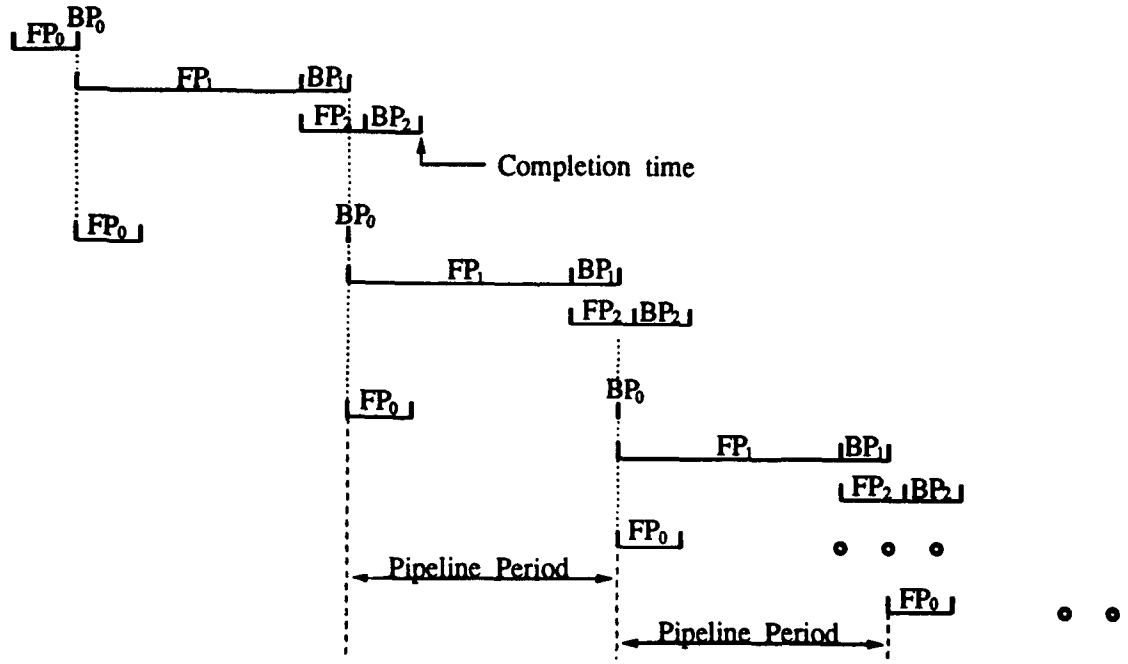


Figure 21: Completion time and throughput analysis of the simple system.

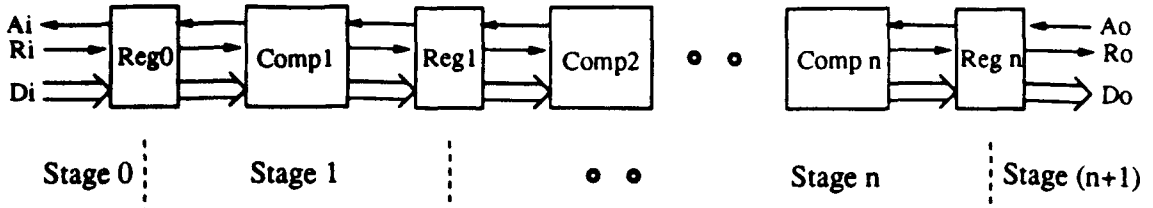


Figure 22: A linear pipelined system.

to $\frac{1}{(FP_i+BP_i)}$. Since two consecutive stages i and $(i+1)$ have a joint event Ai(Reg$i$), the throughput rate of these two stages, which equals the firing rate of event Ai(Reg$i$), is less than or equal to $\frac{1}{(FP_i+BP_i)}$ and $\frac{1}{(FP_{(i+1)}+BP_{(i+1)})}$. Therefore, the pipeline period of these two stages i and $(i+1)$ is greater than or equal to (FP_i+BP_i) and $(FP_{(i+1)}+BP_{(i+1)})$; the lower bound of the system pipeline period equals the maximum of (FP_i+BP_i) for all stage i. A further observation is from the timing diagram in Figure 21. The data forward execution on a stage is concurrent with the data backward execution on its previous stage, e.g., FP_1 is overlapped with BP_0 in the timing diagram, and FP_2 is overlapped with BP_1. Therefore, the completion time is mainly determined by the the forward propagation delay time.

5.3.2 Performance analysis of a linear pipeline

From previous analysis, a stage is formed by two registers without any register block between them. Without loss of generality, a linear pipeline is defined as a series of computation blocks with a register between any two consecutive computation blocks, as shown in Figure 22. In this figure, registers are labeled Reg0, Reg1, ..., Reg$n$, and the computation block between Reg$(i-1)$ and Reg$i$ is labeled Comp$i$ for $1 \le i \le n$. There are $(n+2)$ stages in the system, and the forward and backward propagation delay times of these stages are defined below.

$$
\begin{aligned}
FP_0 &= D_{sfl}(Reg0) \\
BP_0 &= 0 \\
FP_i &= D_{sp}(Reg(i-1)) + D_{fp}(Compi) + D_{sfl}(Regi), && \text{for } 1 \le i \le n \\
BP_i &= D_{bp}(Compi) + D_{sbl}(Reg(i-1)), && \text{for } 1 \le i \le n \\
FP_{(n+1)} &= D_{sp}(Regn) \\
BP_{(n+1)} &= D_{sbl}(Regn)
\end{aligned}
$$

By generalizing the result of the previous example, the performance measures of this system can be formulated as follows.

$$
\begin{aligned}
L &= \sum_{i=0}^{n+1} FP_i + BP_{(n+1)} + \Delta, \\
&\qquad \text{where } \Delta = \max\{0, \max_{i=n}^{1}\{BP_i - \textstyle\sum_{j=i+1}^{n+1} FP_j - BP_{(n+1)}\}\} && (4) \\
P &= \max_{i=0}^{n+1}\{(FP_i + BP_i)\} && (5) \\
R &= 1/P && (6)
\end{aligned}
$$

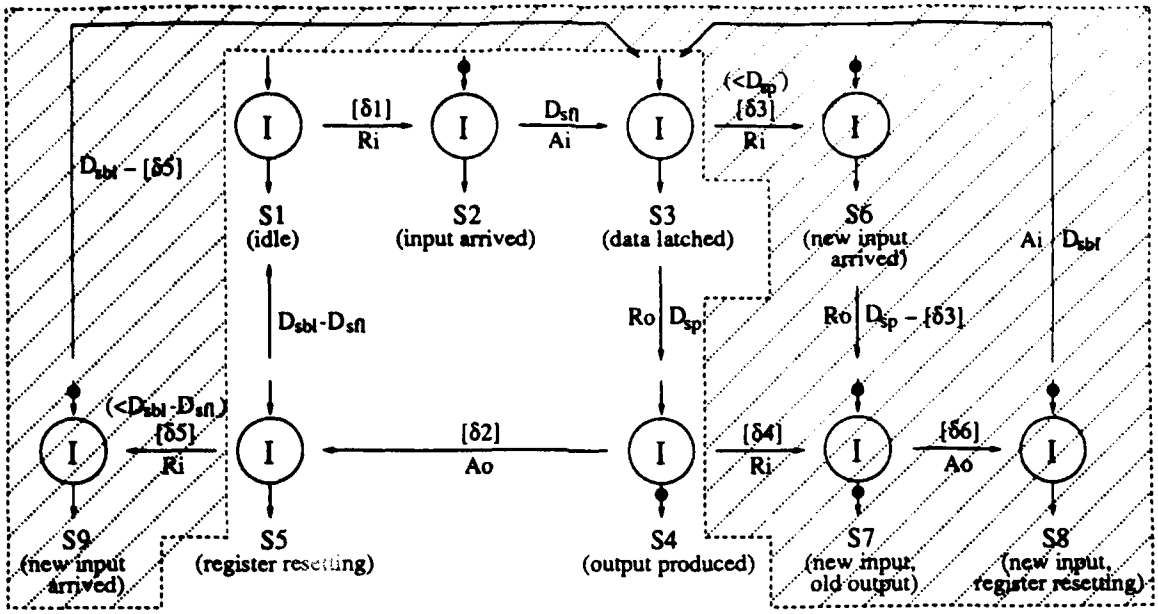


Figure 23: Timing model for storage nodes.

In equation (4), Δ is most likely to be zero because usually $\sum_{j=i+1}^{n+1} FP_j \gg BP_i$. We conveniently assume Δ to be zero.

5.4 Timing Model for DFG/EDFG

After understanding the timing behavior of asynchronous systems at the circuit level, we defined a timing model for the high-level data flow specification which reflects the behavior of the low-level implementation. The key to model timed behavior for data flow specification is the interpretation of tokens in the data flow specification with respect to the events in the block model, i.e., the token-handshaking protocol relation described in Section 3.3. Since an EDFG is an abstract representation of an asynchronous system and a DFG description can be replaced by an EDFG description, we begin with the timing model for EDFG.

5.4.1 Timing Model for EDFG

an EDFG is an abstract representation for an asynchronous system, so the timing parameters, D_{sfl}, D_{sbl}, D_{sp}, D_{fp}, and D_{bp}, are directly attached to corresponding nodes in the EDFG description. For storage nodes, $D_{sfl/sbl}$ and $(D_{sfl/sbl} + D_{sp})$ correspond to input token absorption time and the time of moving a token from input to output; the usage of D_{sfl} or D_{sbl} in the above delay time calculation depends on the state of the storage node when the input token arrives. The timed state diagram for a storage node is shown in Figure 23, where each state corresponds to a possible token distribution in Figure 18(a). Two labels are attached to each directed arc between two states in Figure 23: the delay between the two states and the event which occurs between two states. For example, S1 in Figure 23

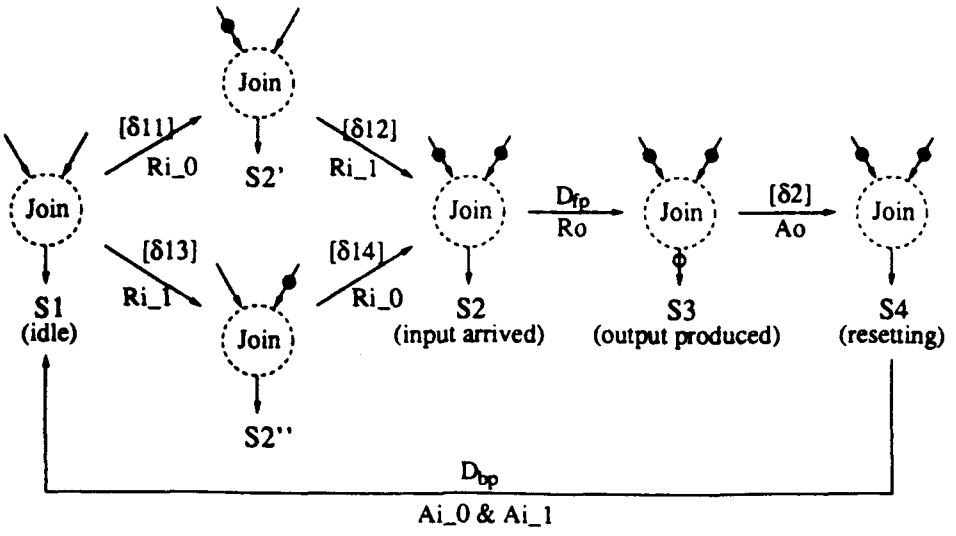


Figure 24: Timing model for non-storage nodes.

corresponds to the initial token distribution shown in Figure 18(a). After event Ri occurs, S1 transits to S2 and the time interval between S1 and S2 is $\delta 1$. S2 in Figure 23 corresponds to the token distribution of the post-Ri condition and the post-Ao condition in Figure 18(a). The unshaded part of Figure 23 represents the arrival of input token at the state that the register does not hold another data, and D_{sfl} is the latch time for the token absorption; the shaded part represents the arrival of input token at the state that the register is holding another data, and D_{sbl} is the latch time for the token absorption. $\delta 1$, $\delta 2$, $\delta 3$, $\delta 4$, $\delta 5$, and $\delta 6$ are delays associated with environment, where $\delta 1$, $\delta 3$, $\delta 4$, and $\delta 5$ represent the input token arrival time and where $\delta 2$ and $\delta 6$ represent the output token removal time.

For phantom nodes, D_{fp} represents the time from the generation of an input token to the generation of the corresponding output token, and D_{bp} represents the time from the removal of an output token to the removal of the corresponding input token. In order to allow the reader to understand that multi-input/multi-output phantom nodes share the same behavior of single-input single-output phantom nodes, we present a timed state diagram for a two-input MJoin in Figure 24, where each state corresponds to a possible token distribution in Figure 18(b). For example, S1 in Figure 24 corresponds to the initial token distribution shown in Figure 18(b). After both Ri_0 and Ri_1 occur, S1 transits to S2. S2 in Figure 24 corresponds to the token distribution of the post-Ri condition in Figure 18(b).

The timing model described in this section is an enhancement for the (extended) data flow model in Section 3.3 and Section 4.3.2. Comparing the unshaded part of Figure 23 with Figure 6(b) for the storage node, S3 is an extra state in the timed extended data flow model, and S3, which is between S2 and S4, describes the transient states in Figure 6(a); S5, which represents the register resetting state after the removal of the output token, also is an extra state in the timed model, and it is merged in the idle state ub Figure 6(a). Comparing

Figure 24 with Figure 12(b) for the phantom node. S4 is an extra state in the timed extended data flow model. Again, S4 describes the transient state in Figure 12(a). At this point, the timed extended data flow model has fully reflected the low-level behavior in the high-level description.

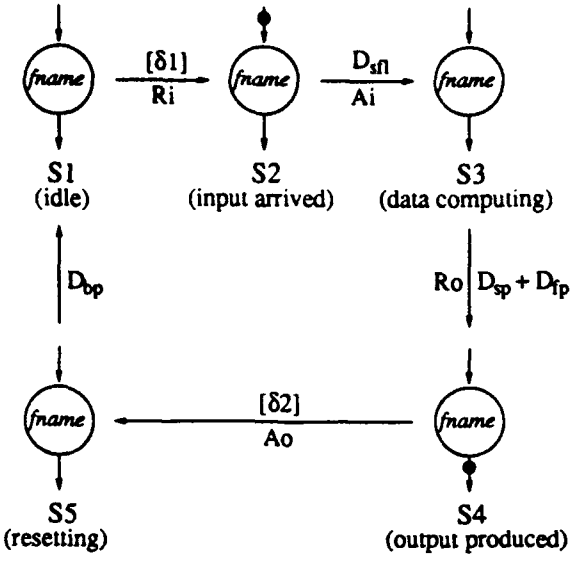


Figure 25: Timing model for DFG nodes.[7]

5.4.2 Timing Model for DFG

Each node in a DFG corresponds to an EDFG phantom node plus a storage at each input of this EDFG node (see Figure 15). Therefore, we can simply use the timing model of the EDFG to simulate a DFG. On the other hand, we can develop a timing model for the DFG by using the timed Petri net model of the composition of the register block and the non-register block in Figure 19. The timed state diagram for a DFG node is shown in Figure 25. Comparing it with Figure 6(b), there are two extra states in Figure 25, where S3 corresponds to the transient states for the data latch and the data computation, and where S5 corresponds to the transient state for functional unit resetting. Based on the analysis in Section 5.3.2, we further simplify the timing model shown in Figure 26, where we need only two timing parameters, D_{FP} and D_{BP}. Referring to Section 5.3.2, D''_{sfl} in D_{FP} is the forward latch time of the output register of the function unit, and D'_{sbl} in D_{BP} and D'_{sp} in D_{FP} are the backward latch time and the propagation delay time of the input register of the function unit. In other words, we adopt the stage delay parameters to the simplified model. This simplified DFG timing model complys with the data flow model in Section 3.3, and it reduces the simulation complexity, as well as provides a simpler model for high-level

[7]The part corresponding to the shaded part in Figure 23 is not shown in this model.

synthesis problems.

According to the timed behavior of a DFG/EDFG, we can simulate and analyze a system from the behavior description. Furthermore, the attached parameters and analysis formulations can be used as measures in a high-level synthesis. These will be discussed in the next section.

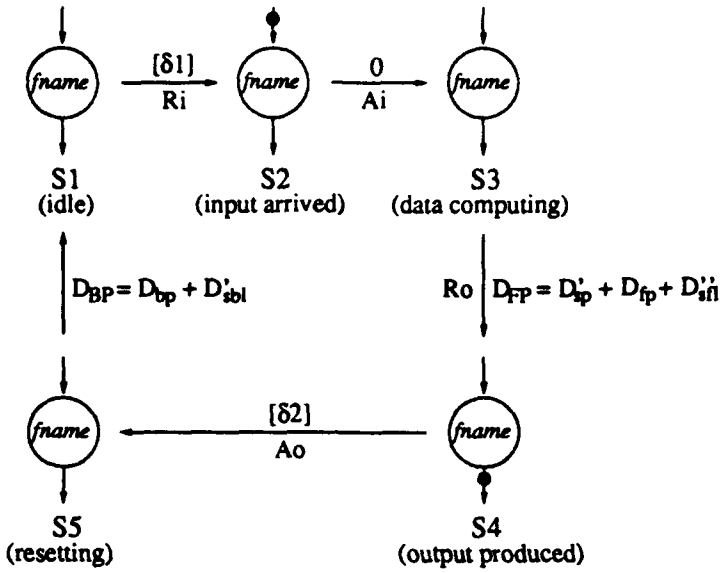


Figure 26: Simplified timing model for DFG nodes.

6 Data Flow Specification for High-Level Synthesis

One of main reasons to use a data flow specification is to have a sound functional and timing model to describe system behavior so that designers can make design decisions at the high level. In this section we first discuss the transformations which preserve the functionality of DFG descriptions and result in different realizations with different performances. Two kinds of function-preserved transformations are discussed: *sharing schemes* which provide design templates to map a DFG to another DFG with fewer atomic function nodes; and *local transformations* which are rules to map a substructure of a DFG to another substructure so that the area and/or the performance of the new DFG is improved. Based on the sharing schemes in this section and on the performance model described in the previous section, we address the sequencing and allocation problem in which we want to find sequences and allocations of atomic functions (of data paths) with optimized or near-optimized system performance and/or area consumption.

6.1 Sharing Schemes

Resource sharing is broadly used to reduce the hardware size as well as the implementation cost in digital system designs. There are two issues related to resource sharing in the design process:

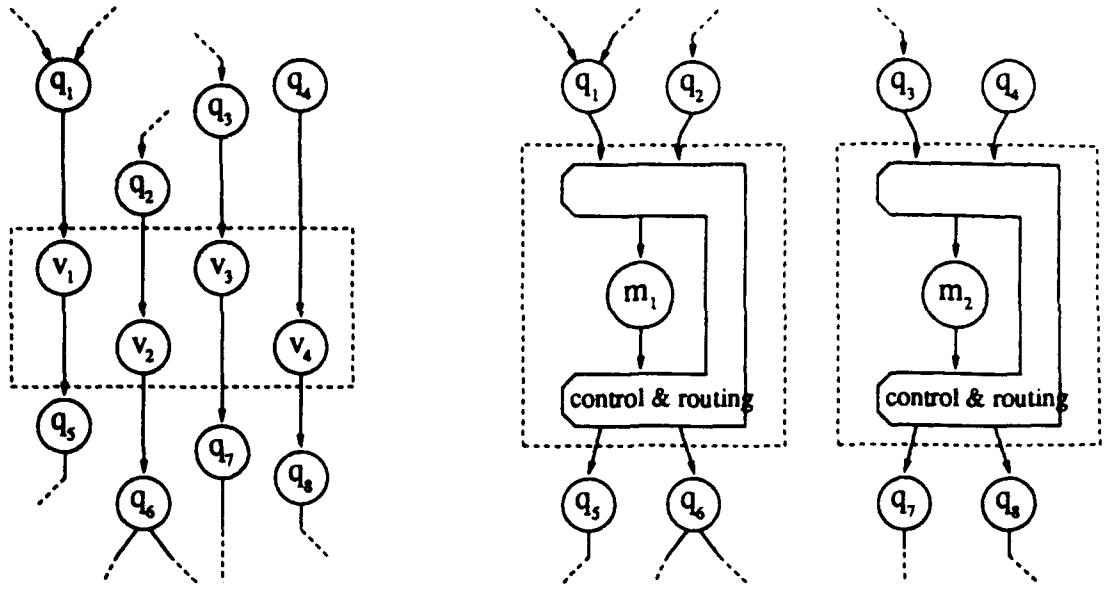
1. Allocation – For each operator, what are those operations going to be executed by the shared operator?

2. Sequencing – For each operator, what is the execution order of those operations which share the operator?

Here we focus on the transformations of a DFG into another DFG for a given sequence and allocation of operations under the DFG paradigm, namely, sharing schemes.

In our synthesis system, each node in a DFG is implemented by a hardware module, i.e., an operator. Assume that there is no multi-function operator such as an ALU. Therefore, a sharing scheme will transform an (original) DFG into a (mapped) DFG in the following way: the same type of N atomic function nodes of the original DFG is replaced by M same type of nodes with proper control and routing structure in the mapped DFG, where $N > M$; the sequence of operations in the original DFG is preserved by the sequence of operations in the mapped DFG, i.e., system functionality is preserved. There are three parts in each sharing scheme: the *shared unit(s)*, the *control part*, and the *data routing part*, which are together referred to as the *sharing structure*. Figure 27 demonstrates an abstract sharing scheme for $N = 4$ and $M = 2$. In the mapped DFG, the output of $q_1(q_2)$ is executed by m_1, and this execution generates an output as the input of $q_5(q_6)$; the output of $q_3(q_4)$ is executed by m_2, and this execution generates an output as the input of $q_7(q_8)$. The control part and the routing part in the sharing structure are used to ensure that the sequence of operations in the mapped DFG preserves the sequence of operations in the original DFG. Therefore, the mapped DFG uses fewer functional nodes than the original one, while performing the same functions; however, the mapped DFG has extra control/routing nodes.

Classification and notation Let the same type of N atomic function nodes in the original DFG be labeled $v_1, v_2, \ldots, v_N$, and the M same type of nodes in the mapped DFG be labeled $m_1, m_2, \ldots, m_M$. There are two classes of sharing schemes: sharing schemes with fixed allocation and sharing schemes with dynamic allocation. In sharing schemes with fixed allocation, the execution of each node v_i is assigned to a specific node m_j. Let GN_j be the set of v's assigned to m_j, i.e., there are $|GN_j|$ operations sharing operator m_j. On the other hand, the execution of each node v_i in sharing schemes with dynamic allocation is not assigned to a specific node m_j. Instead, the execution of node v_i is dynamically assigned to any operator m_j, which usually is an idle one at the time when the input data of node v_i is available. Let $GN_{j_1,j_2,\ldots,j_k}$ be the set of vs to be dynamically allocated to the set of k operators $m_{j_1}, m_{j_2}, \ldots, m_{j_k}$, where $|GN_{j_1,j_2,\ldots,j_k}| > k$, i.e., $|GN_{j_1,j_2,\ldots,j_k}|$ operations sharing the k operators. Without loss of generality, the presentation of following schemes will use $|GN_j| = 4$ for sharing schemes with fixed allocation and $|GN_{j_1,j_2,\ldots,j_k}| = 4$ and $k = 2$ for sharing schemes with dynamic allocation. The original four operations for $|GN_j| = 4$ or



(a) Before applying sharing scheme (b) After applying sharing scheme

Figure 27: An abstract sharing scheme with fixed allocation for $N = 4$, $M = 2$.

$|GN_{j1,j2}| = 4$ are shown in Figure 28, where I1, ..., I4 can be either connected to input port(s) of the original DFG or connected to output(s) of some nodes in the DFG, and O1, ..., O4 can be either connected to output port(s) of the DFG or connected to input(s) of some nodes in the DFG. For example, one DFG has the four operations connected serially, e.g., O1, O2, and O3 are connected to I2, I3, and I4 respectively, and I1 and O4 are connected to the input port and the output port of the DFG.

6.1.1 Sharing Schemes with Fixed Allocation

The problem in a fixed allocation is to map a set of operations into a shared unit (operator). Figure 27 shows the outline of this kind of sharing scheme. Besides the shared unit, the control part generates condition tokens to control the data routing part so that these operations are executed by the shared unit in a certain order. Two ordering schemes for fixed allocation are presented. One is the scheme with variable sequence, in which the order of operations processed by a sharing structure is based on the first-come-first-served (FCFS) ordering scheme. The other is the scheme with fixed sequence, in which the order of operations processed by a sharing structure is pre-determined by system designers or scheduling algorithms during the process of high-level synthesis.

Fixed-allocation sharing scheme with variable sequence Figure 29(a) is a fixed-allocation sharing scheme with variable sequence for $|GN_j| = 4$. In this scheme, an MSelector and an MDistributor with condition input form the data routing part. Four $R([\,])$ functions

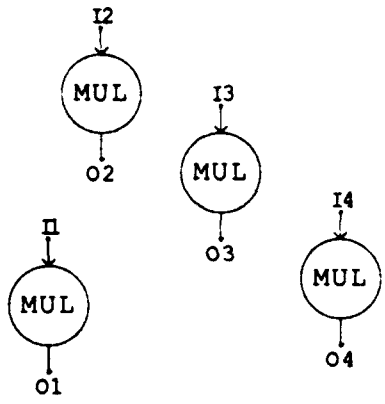


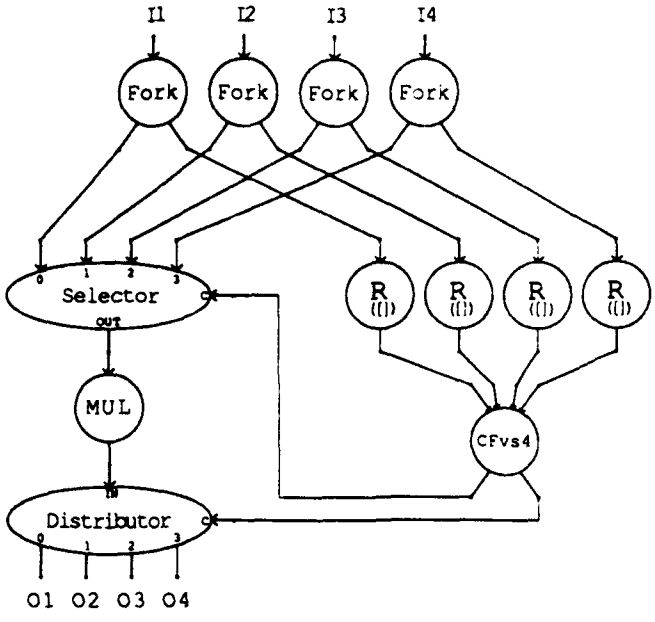
Figure 28: Four operations for $|GN| = 4$.

and a CFvs4 function form the control part. $R([\,])$ is an atomic function which passes the input token to the output without passing the input data value. CFvs4[8] is a macro function which generates condition tokens to indicate which input token is available so that a proper data path in the data routing part is open. Figure 29(b) is the DFG definition of function CFvs4, where C(⟨const⟩) is an atomic function which generates a token with constant data value ⟨const⟩ if it receives a null data token. For example, I2 has data available, and it activates the second $R([\,])$ to generate a null data token to CFvs4. Due to the token from the second input of CFvs4, "$'b01$" [9] data tokens are generated on both outputs of CFvs4, and they will open the data route from input 1 of the MSelector to the output 1 of the MDistributor. In case there is more than one input data available, an Arbiter in CFvs4 will decide the order of routing paths/operations based on an FCFS ordering scheme.

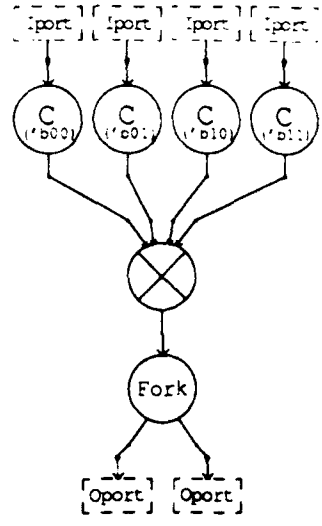
Fixed-allocation sharing scheme with fixed sequence Figure 30(a) is a fixed-allocation sharing scheme with fixed sequence for $|GN_j| = 4$, and it is similar to the sharing scheme in Figure 29(a) except for the control part. CFfs4, which forms the control part of Figure 30(a), is a macro function which generates fixed sequence of condition tokens so that data paths are open in a pre-determined order. Figure 30(b) is the DFG definition of function CFfs4, where COUNT4 is an atomic function which generates the next condition token from the current condition token. In this example, CFfs4 is a two-bit cyclic counter with the initial value "$'b00$", i.e., it generates the sequence of conditional tokens, "$'b00$", "$'b01$", "$'b10$", and "$'b11$" repeatedly. Therefore, the order of operations is MUL(I1) followed by MUL(I2) followed by MUL(I3) followed by MUL(I4). Even though the data at I4 is available before the data I3, the data at I4 cannot be processed until MUL(I3) is completed. The order of operations can be changed either by changing the input/output location or by changing the sequence generator.

[8] CFvs stands for the control function with variable sequence.

[9] $'b$, $'o$, and $'h$ are used to lead the binary, octal, and hexadecimal numbers respectively.

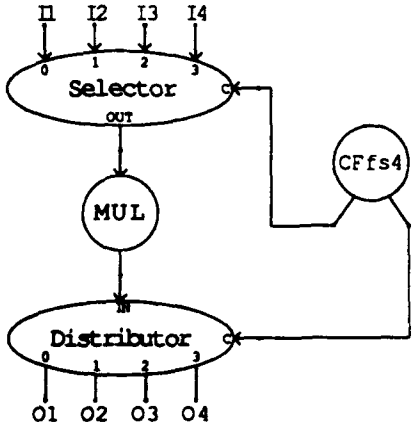


(a) Sharing scheme with fixed allocation with variable sequence

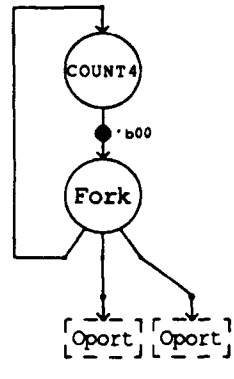


(b) Definition of CFvs4

Figure 29: A sharing scheme with fixed allocation with variable sequence.



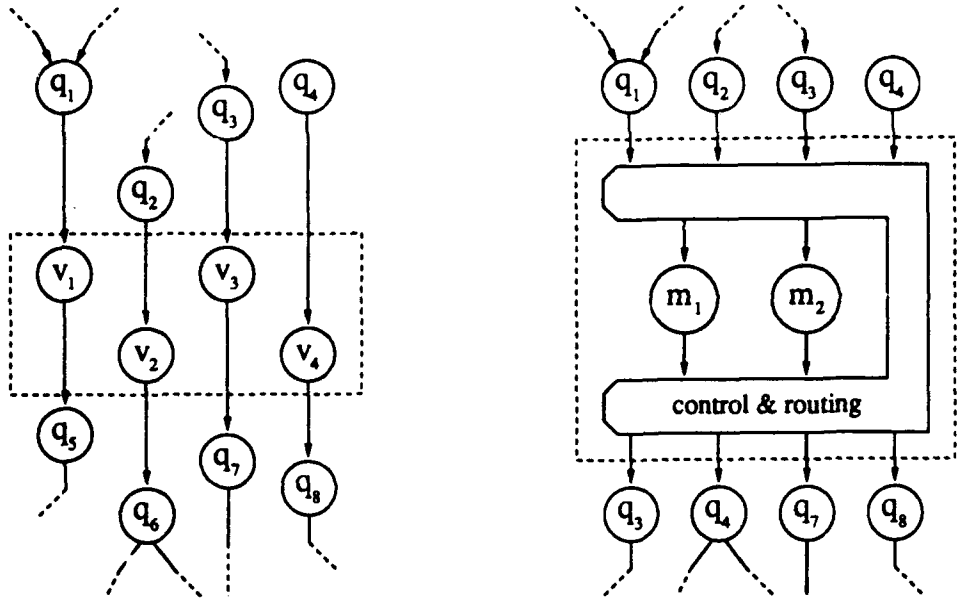
(a) Sharing scheme with fixed allocation with fixed sequence



(b) Definition of CFfs4

Figure 30: A sharing scheme with fixed allocation with fixed sequence.

Functionality preservation In order to show that the mapped DFG preserves functionality, we need to show that the order of operation execution in the original DFG is preserved by the mapped DFG. In an FCFS ordering scheme, no specific order enforces the operations which share the same operator. In fact, any order of these operations is acceptable to the sharing structure. Therefore, the mapped DFG which uses the FCFS ordering scheme follows and preserves the operation execution order of the original DFG. On the other hand, the sharing scheme with fixed sequence may have a problem preserving the operation execution order. For example, I1 is connected to O2 in the original DFG, so I1 has to be executed after I2 is executed to generate output O2. Therefore, the sequence provided by Figure 30(b) would not work. In order to use the sharing scheme with fixed sequence, designers and synthesis algorithms need to give a sequence which preserves original operation ordering, i.e., *data dependency*. There are many ways to solve this problem, e.g., interchanging I1, I2 and interchanging O1, O2 will fix the problem in Figure 30.



(a) Before applying sharing scheme (b) After applying sharing scheme

Figure 31: A abstract sharing scheme with dynamic allocation for $N = 4$, $M = 2$.

6.1.2 Sharing Scheme with Dynamic Allocation

The problem in this kind of sharing scheme is to map a set of operations into a set of shared units (operators) dynamically. Figure 31 shows the outline of this kind of sharing scheme. In this sharing scheme, the data routing part should have routes from every data input to every shared unit as well as routes from every shared unit to every data output. The condition tokens generated by the control part not only need to indicate the input-to-output to be executed, but also need to indicate which shared unit is used. By generalizing the

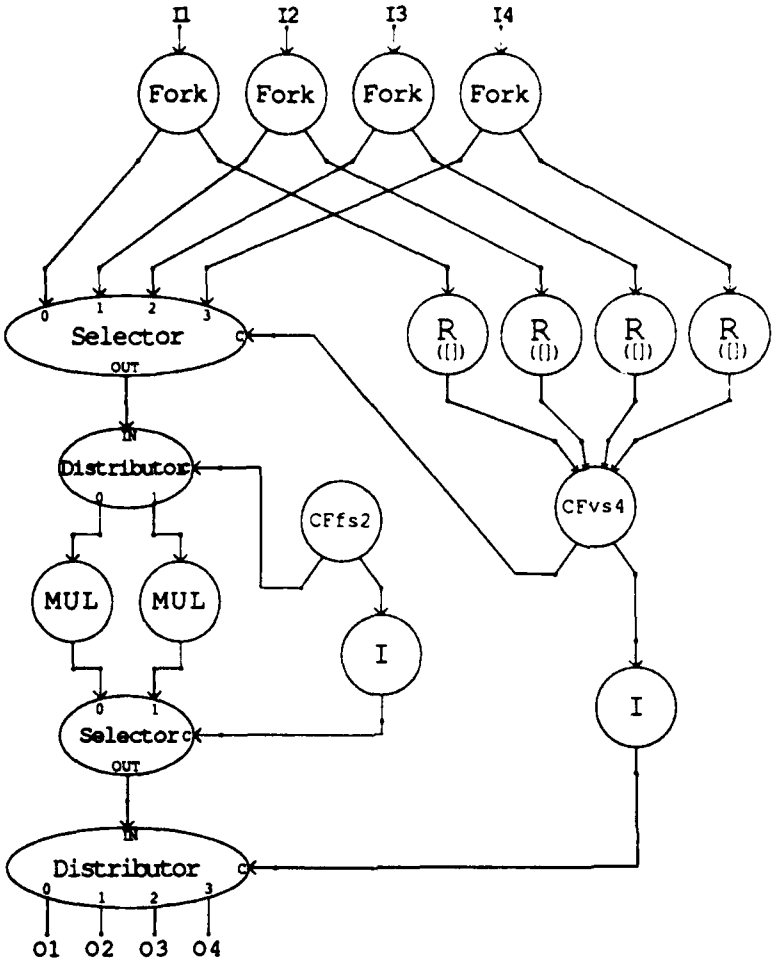


Figure 32: A sharing scheme with dynamic allocation.

operation ordering schemes and creating different operator allocation schemes, there may be many different kinds of sharing schemes under this category. Here we only present one sharing scheme. In this sharing scheme, the FCFS ordering scheme is used for the operation ordering. By assuming that the first started operator is first released, the sequentially cyclic ordering is used for the operator allocation, i.e., operator 1 to operator k is sequentially and cyclically allocated. Figure 32 is a sharing scheme with dynamic allocation of this kind for $|GN_{j_1,j_2}| = 4$. In this sharing scheme, the four-input MSelector and the four-output MDistributor, which control the input-output routing, and the two-output MDistributor and the two-input MSelector, which control the operator routing, form the data routing part. Four $R([\ ])$ functions, function CFvs4, which generate tokens to control the FCFS input-output routing, and a 1-bit counter CFfs2, which generates tokens to control the operator routing, form the control part. I is an atomic function which passes input token to output, and it is used to store a condition token for an unfinished operation in the sharing structure,

e.g., there may exist two sets of condition tokens for two operations in the structure at the same time in Figure 32. For example, I2 has data available first, and it activates the second $R([\,])$ and makes CFvs4 generate "$'b01$" data tokens to the condition port of the four-input MSelector and to the condition port of the four-output MDistributor. Meanwhile, CFfs2 generates "$'b0$" to the condition port of the two-input MDistributor and to the condition port of the two-output MSelector. Then the data I2 is passed to the input of the left MUL in Figure 32. If I4 has data available, right after I2 has data available. Because one *1* at the output of CFvs4 keeps the condition token "$'b01$" for I2, CFvs4 can generate new "$'b11$" tokens for I4. Similarly, CFfs2 can generate "$'b1$" after I2 is passed through the two-input MDistributor of two MULs. Therefore, data at I4 can start being executed by the right MUL even though the data of I2 is still in the sharing structure.

6.1.3 Sharing Scheme with Micropipelined Shared Unit

A node/function which is *micropipelined* [28] by being partitioned into a pipeline, i.e., this node becomes a macro function defined by a series of sub-functions. For example, MUL

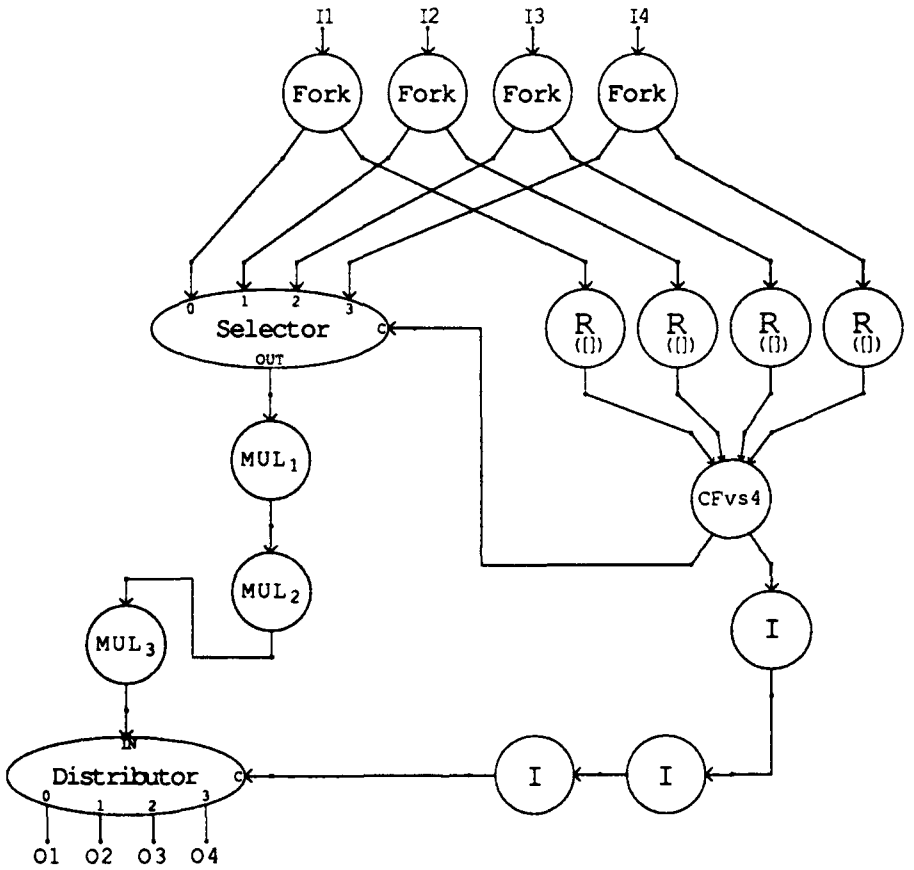


Figure 33: A sharing scheme with micropipelined shared units.

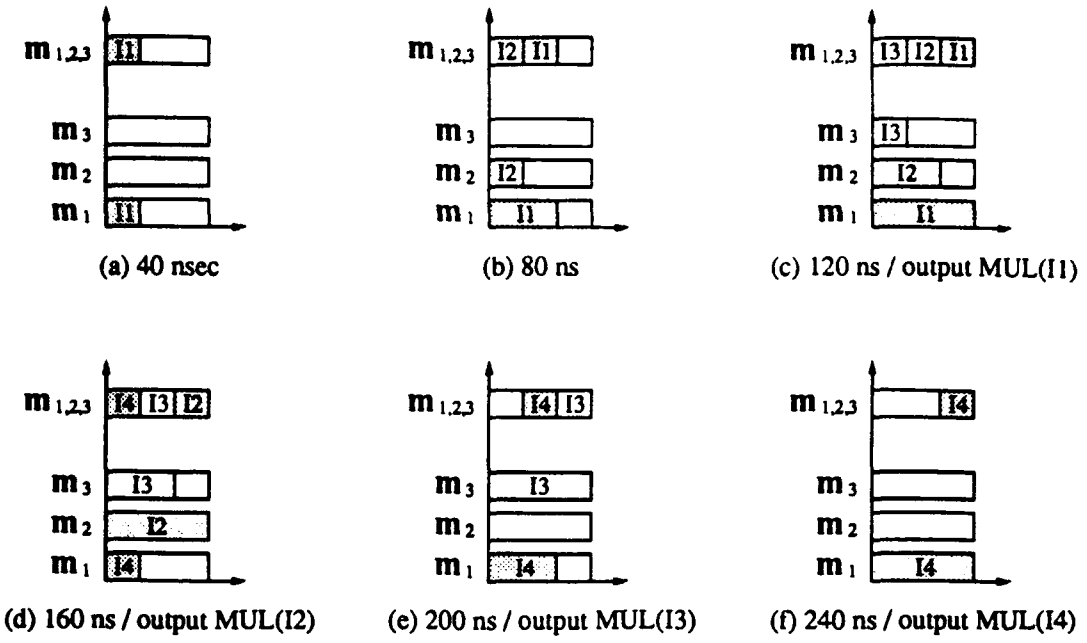


Figure 34: Timing behavior of micropipelined sharing scheme.

may be partitioned into three sub-functions, MUL_1, MUL_2, and MUL_3, and $MUL(i)$ equals $MUL_3(MUL_2(MUL_1((i))))$ for each input i, which includes two operands. By using micropipelined shared units, there are still two classes of allocation methods as stated previously. However, we may use a fixed allocation sharing scheme with a micropipelined function unit, whose behavior resembles the behavior of a sharing scheme with dynamic allocation with multiple shared units. Figure 33 shows a sharing scheme with a three-stage micropipelined shared unit, and it is a fixed allocation sharing scheme with FCFS ordering scheme for $N = 4$ and $M = 1$. The behavior of Figure 33 is similar to the behavior of Figure 32 with $N = 4$ and $M = 3$. Without considering the control overhead, the timing and resource usage diagram in Figure 34 shows the behavioral resemblance between these two schemes, where $m_{1,2,3}$ represents three stages of the micropipelined shared unit, the execution time for each stage is 40 nsec, m_1, m_2, and m_3 represent three shared units, and four operations are executed in the order of I1, I2, I3, and I4.

6.1.4 Effects of Sharing Schemes

Since a sharing scheme is a fixed template with respect to the number of sharing operations, the number of shared operators, and the kind of scheme, we can easily estimate the effects of the sharing scheme such as area and performance.

Area Although a sharing scheme maps one DFG into another DFG with fewer atomic function nodes, some extra nodes are required to form the routing part and the control part in the mapped DFG. Therefore, the area gain of a mapping equals the area of the eliminated

atomic functions; the area overhead of the mapping equals the area of these extra nodes. Because resource sharing always reduces system performance, if the area overhead is greater than the area gain, then this mapping should be abandoned.

Performance After sharing schemes are applied, the execution time of each operation which shares an operator with other operations is increased by the routing delay and possible control delay. Besides the overhead caused by the sharing structure, the starting execution time of some operations is postponed due to the sequence enforced by the sharing structure.

The performance and area effects of DFG mapping can be easily obtained by simulating the mapped DFG and by counting the area of the mapped DFG. Furthermore, we need to quantify these effects in terms of the kind of sharing scheme and the number of sharing operations and shared operators, so that these quantities can be used in high-level design decisions such as sequencing and allocation problem.

Example [effects of sharing schemes with fixed allocation having fixed sequence] Let us analyze the effects of sharing schemes shown in Figure 30. Assume that there are N operations which share one operator, where $N > 1$. Let the area cost of the operator for the operation be A_{op}; let the area cost of the n-input MSelector (n-output MDistributor) be $n * A_s$ ($n * A_d$) with the constant factor A_s (A_d); and let the area cost of the n-bit counter be $n * A_c$ with the constant factor A_c. The area gain/overhead analysis for the sharing scheme is shown as follows:

Area gain:	$(N-1)$ operators;
Area overhead:	an N-input MSelector,
	an N-input MDistributor, and
	a $\lceil log_2 N \rceil$-bit counter;
Total area gain:	$(N-1) * A_{op} - N * (A_s + A_d) - \lceil log_2 N \rceil * A_c$

Based on the analysis of our design library, the forward propagation delay time of the n-input MSelector, $FP_{MSelector}(n)$, can be formulated as $k_1 + \lceil log_2 n \rceil * k_2$, where k_1, k_2 are constant factors and $k_1 \gg k_2$ [33]. Generally we can ignore k_2, and assume that $FP_{MSelector}(n) = FP_{MSelector}$ is a constant. The backward propagation delay time of the n-input MSelector, $BP_{MSelector}(n)$, and the forward propagation delay time of the n-output MDistributor, $FP_{MDistributor}(n)$, and the backward propagation delay time of the n-output MDistributor, $BP_{MDistributor}(n)$, can be similarly formulated, so they are assumed to be constants $BP_{MSelector}$, $FP_{MDistributor}$, and $BP_{MDistributor}$. The performance overhead analysis for the sharing scheme is shown as follows:

Overhead of forward propagation delay time: $FP_{ov} = FP_{MSelector} + FP_{MDistributor}$
Overhead of backward propagation delay time: $BP_{ov} = BP_{MSelector} + BP_{MDistributor}$

Furthermore, $FP_{MSelector}$ and $FP_{MDistributor}$ can be assumed to be zero in the sharing structure due to the parallelling of the data computation and the control generation, which is a hardware implementation issue and is not discussed in this report.

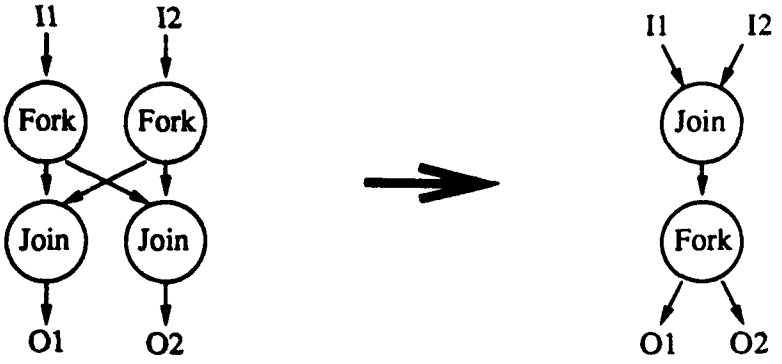


Figure 35: Local Transformation 1.

6.2 Local Transformations

Algorithmic transformations can be used to improve the design efficiency at the behavioral level so that the resulting design description can generate a suitable implementation [27, 29, 31]. Most transformations use the peephole optimization technique, used similarly in the compiler design, and are therefore called *local transformations* in this context. The biases in behavioral level descriptions are caused by the designers' coding style or generated by other transformations such as sharing schemes. Transformations are developed to reduce the number of operations, to reduce the size of control structures, to reduce length of the critical path, to remove the redundancy, and so on. Snow has systematically developed transformations for the C-MU RT-CAD system [27]. Most of Snow's transformations are general enough to have analogous transformations in our system such as *dead activity elimination*, *redundant activity elimination*, *select factoring/combination*, etc., so we will not describe the available transformations in our system. Instead, we will present a few transformations to show the role of tokens and data types in DFG/EDFG transformations. Because of the token model, the correctness of these transformations can be easily proven by symbolic (token) simulation. Later we present one transformation which is often used to reduce the routing part and the control part of sharing structures, and it will be extensively used in the examples of the next section.

Symbolic token simulation Figure 35 is a simple local transformation. We can show that these two DFGs are equivalent by simulation. Given a token with data value d_i and data type t_i for input Ii for $i = 1, 2$ to both DFGs, a token is produced with data value $\langle d_1, d_2 \rangle$ and data type $\langle t_1, t_2 \rangle$ at each of outputs O1 and O2 for both DFGs. Therefore, they are functionally and behaviorally equivalent.

Data type matching In the transformations of DFG, we need to consider not only the equivalence of token generation but also the equivalence of the token value and the token type. Figures 36(a) and (b) are two DFGs, which look equivalent. We can show these two DFGs are not equivalent by simulation. Given a token with data value d_i and data type t_i for

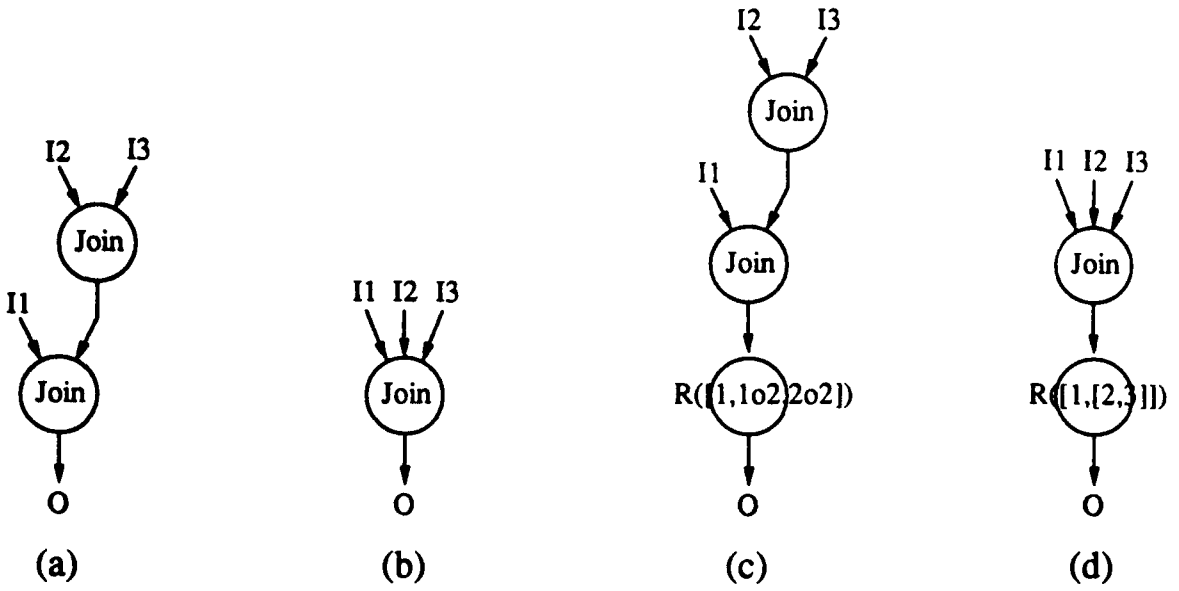


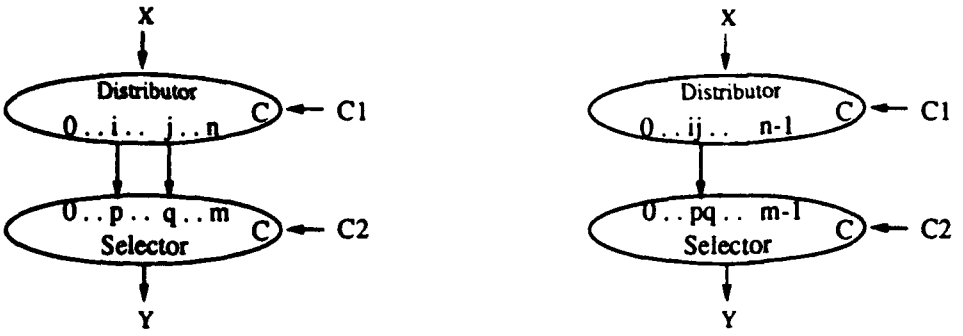
Figure 36: Local Transformation 2.

input Ii for $i = 1, 2, 3$ to both DFGs, a token is produced with data value $\langle d_1, \langle d_2, d_3 \rangle\rangle$ and data type $\langle t_1, \langle t_2, t_3 \rangle\rangle$ at the output of Figure 36(a), and a token is produced with data value $\langle d_1, d_2, d_3 \rangle$ and data type $\langle t_1, t_2, t_3 \rangle$ at the output of Figure 36(a). They are the same in terms of hardware implementation, but they are not the same in terms of the DFG specification. With atomic functions for data type conversion, Figures 36(b) and (c) are equivalent, and Figures 36(a) and (d) are equivalent. $R([1, 1 \circ 2, 2 \circ 2])$ and $R([1, [2, 3]])$ are atomic functions called *routers* in our system. Router functions rearrange input data by copying, repeating, and shuffling input data. The notation of router functions is based on Backus' FP [2], where i is the FP selector function, square bracket $[\ldots]$ is the functional form of construction, and circle $\circ$ is the functional form of composition. These are defined as follows:

$$
\begin{aligned}
i : \langle x_1, x_2, \ldots, x_n \rangle &= x_i, \qquad \text{for any positive } i \leq n; \\
[f_1, f_2, \ldots, f_n] : x &= \langle f_1 : x, f_2 : x, \ldots, f_n : x \rangle; \\
f \circ g : x &= f : (g : x),
\end{aligned}
$$

where x and $\langle x_1, x_2, \ldots, x_n \rangle$ are input objects of functions, and f_1, f_2, ..., f_n, f, and g are functions, e.g., FP selector functions. Therefore, we can show that Figures 36(b) and (c) are equivalent as follows.

$$
\begin{aligned}
[1, 1 \circ 2, 2 \circ 2] : \langle d_1, \langle d_2, d_3 \rangle\rangle &= \langle 1 : \langle d_1, \langle d_2, d_3 \rangle\rangle, \\
&\quad\ 1 \circ 2 : \langle d_1, \langle d_2, d_3 \rangle\rangle, 2 \circ 2 : \langle d_1, \langle d_2, d_3 \rangle\rangle \\
&= \langle d_1, 1 : \langle d_2, d_3 \rangle, 2 : \langle d_2, d_3 \rangle\rangle \\
&= \langle d_1, d_2, d_3 \rangle
\end{aligned}
$$



(a) Before transformation (b) After transformation

Figure 37: Local Transformation 3.

Similarly, we can show that Figures 36(a) and (d) are equivalent, i.e.,

$$[1,[2,3]] : \langle d_1, d_2, d_3 \rangle = \langle d_1, \langle d_2, d_3 \rangle \rangle$$

A transformation for sharing scheme reduction Figure 37(a) is a DFG under a special condition, which often appears after sharing schemes are applied. In this figure, two output ports of an MDistributor are connected to two input ports of an MSelector, and the order of tokens which pass from the input of the MDistributor to the output of the MSelector are preserved, e.g., in Figure 37(a), j obtains a token after i obtains a token from X, then Y obtains a token from q after Y obtains a token from p. In this situation, we can combine these two paths into one with the MDistributor reducing one output port and the MSelector reducing one input port. In addition to reductions these ports, the control token generation for the MDistributor and the MSelector needs to be changed for the transformed DFG. Figure 37(a) is transformed to Figure 37(b) with one possible mapping for the outputs of the MDistributor and the input of the MSelector as shown below.

Output port index mapping for the MDistributor with $ij = i$:

Before transformation	After transformation	
x	x,	if $0 \leq x < i$;
	i,	if $x = i$ or $x = j$;
	$x-1$,	if $i < x \leq n$ and $x \neq j$.

Input port index mapping for the MSelector with $pq = p$:

Before transformation	After transformation	
y	y,	if $0 \leq y < p$;
	p,	if $y = p$ or $y = q$;
	$y-1$,	if $p < y \leq m$ and $y \neq q$.

6.3 Sequencing and Allocation

Among the many different sharing schemes previously mentioned, the sharing scheme with fixed allocation and fixed sequence is often used in digital system design. However, a designer needs to determine not only the allocation of operations but also the execution order of operations, and this determination will have a significant impact on the performance and area of the final implementation. This problem is analogous to the scheduling and allocation problem in synchronous system synthesis, but with no clock-controlled time step, i.e., the scheduling problem in an asynchronous system cannot be viewed as a partitioning of operations into time steps as in synchronous systems [11, 18]. This problem is closely related to the *resource-constrained project scheduling problem* [3], and the temporal aspects of this scheduling problem can be equivalently represented by partial orders [22]. Therefore, this problem is called the *sequencing and allocation problem* in asynchronous system synthesis.

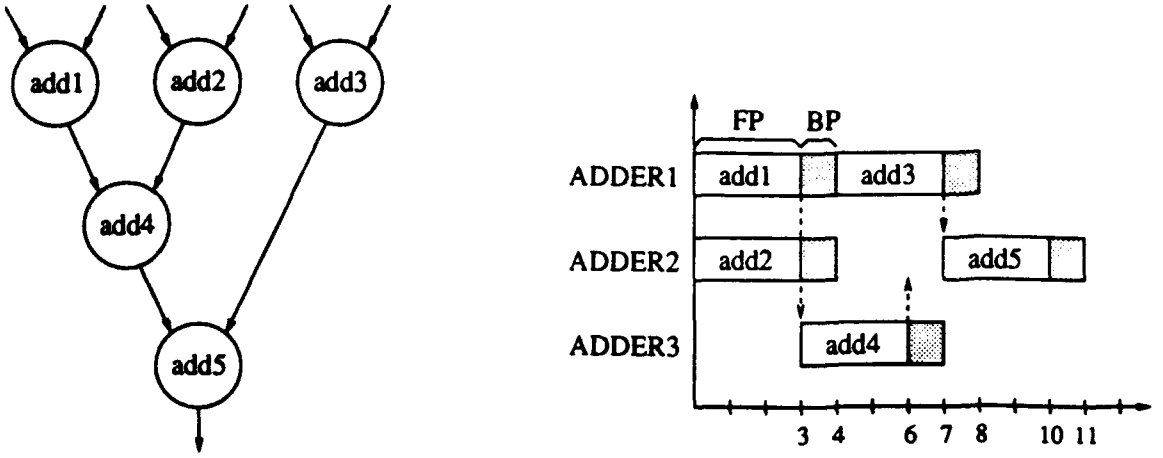
Based on the goal of the synthesis task, there are many kinds of synthesis problems, e.g., the *cost-constrained synthesis problem* and the *performance-constrained synthesis problem* [24]. In this report, we only formulate and provide algorithms for the *resource-constrained sequencing and allocation problem* for non-pipelined systems.

6.3.1 Problem Statement

The problem that we are addressing is how to sequence and allocate operations of an asynchronous system for a given set of resources (operators) so that the system will perform efficiently. The behavior of an asynchronous system is described by a DFG. There are a set of n operations in a system, each of which corresponds to an atomic function in the DFG and belongs to a specific type. Data precedence among operations is implied by the DFG description, where each directed arc represents the direction of data to be transferred between two operations. There are k types of operations in the system. For each type of operation, there is at least one resource operator. Each operator is associated with a computation delay time and a backward control delay time, which correspond to the DFG timing parameters D_{FP} and D_{BP} respectively. The operators of the same type don't have to be identical, i.e., they don't have to have the same values of D_{FP} and D_{BP}. For a non-pipelined system, the system performance is determined by the completion time. (For a pipelined system, the system performance is determined by the throughput rate or the pipeline period.) Our synthesis problem of a system involves following tasks:

- What operator is each operation allocated to?
- What is the execution order of these operations which are allocated to the same operator?

The objective of the problem is to minimize the system completion time. Currently our synthesis algorithms assume that the DFG is acyclic, so the user needs to unroll the loops or choose the loop body before synthesizing the system.



(a) A five-addition DFG (b) The Gantt chart of a sequence and allocation

Figure 38: A simple example of the sequence and allocation problem.

6.3.2 Timing Model for Sequencing and Allocation Problem

The time model used in the sequence and allocation is the same model that is described in Section 5.4.2. Here we want to show how this timing model is used in the sequence and allocation problem and how the sharing effects discussed in section 6.1.4 are used in this timing model for synthesis.

Computation time and resource occupied time In an asynchronous system the time interval during which an operation is computed by an operator, and the time interval during which the operator is occupied by the operation, are different. In Figure 21, FP_i corresponds to the computation time of an operation at stage i, and $(FP_i + BP_i)$ corresponds to the time that stage i is occupied by the operation for a non-pipelined system operation. The time that stage i is occupied by an operation is greater than $(FP_i + BP_i)$ for a pipelined system operation. E.g., $(FP_0 + BP_0 + \delta)$ is the time interval during which an operation occupies stage 0 for the pipelined operation in Figure 21, where δ is the time that the operation is waiting for the input register of the next stage to be available. Since we are dealing with non-pipelined system operation, we don't need to worry about δ at this time. Figure 38(a) is an example DFG with five additions. There are three identical adders available, ADDER1, ADDER2, and ADDER3, and they have $D_{FP} = 3$ and $D_{BP} = 3$. For convenience, we use a two-input addition function to reduce the analysis complexity and we do not consider the sharing overhead in this example. Figure 38(b) is a Gantt chart, representing a sequence and allocation for these five additions, where dashed arrowed lines represent the data precedence of the DFG. In this example, add4 can start when both add1 and add2 finish their computation because add4 does not share the same operator with either. On the other hand, add3 can start only when add1 releases the ADDER1 because they are both allocated to ADDER1 and add3 is sequenced after add1.

Sharing effects The sharing scheme used in the sequence and allocation in this report is the sharing scheme with fixed allocation and fixed sequence. The sharing effects of this scheme have been analyzed in Section 6.1.4. Since we are dealing with the resource-constrained synthesis task, the area cost is determined when the resource constraints are set. Therefore, the area cost is not considered in this synthesis algorithm. On the other hand, FP_{ov} and BP_{ov} play important roles in the measure of the system performance, and they will inflate the length of the computation delay time D_{FP} and the backward propagation delay time D_{BP} of an operation. The system completion time is determined mainly by the computation delay times of the operations in the system. Fortunately, FP_{ov} can be assumed to be zero, so we can separate the sharing penalty from the system completion time, though BP_{ov} may still have certain effects on the system performance.

6.3.3 Algorithms

In this report, we only present two algorithms to solve the resource-constrained sequencing and allocation problem. The details about theoretical basis of these methods can be found in [3]. They are a heuristic algorithm and a branch and bound algorithm.

Heuristic algorithm This algorithm uses the longest path delay from the output of the node to the output of the DFG to prioritize all operations in the DFG, then schedules these operations one by one to available resources. The delay parameter used for the path delay and the operation delay is the computation delay of each node, while the backward control delay is only used to determine the resource occupation time by an operation. Let S be the prioritized list. Let $T_{start}(v)$, $T_{cmp}(v)$, and $T_{release}(v)$ represent the starting time, the computation completion time, and the resource-released time of each operation v. The algorithm is described below.

1. Determine the critical path delay of the DFG, the longest path delay from the input of the DFG to the input of each node, and the longest path delay from the output of each node to the output of the DFG.

2. Find the operations which only receive data from input ports of the DFG; according to the longest path delay from the output of each node to the output of DFG, sort these nodes into list S in non-increasing order.

3. If S is empty, then exit.
   Let v is the first operation in S.

4. Assign operation v to a resource and schedule the operation to intervals $[T_{start}(v), T_{cmp}(v)]$ and $[T_{cmp}(v), T_{release}(v)]$ such that

   - $T_{start}(v) \geq T_{cmp}(w)$ for every w which is a parent node of v.
   - $T_{cmp}(v) - T_{start}(v)$ equals the computation delay time of the operator(resource).
   - $T_{release} - T_{cmp}(v)$ equals the backward control time of the operator.
   - The resource to be assigned is available during the time $[T_{start}(v), T_{release}(v)]$.

5. v is scheduled, and it is removed from S.

6. Find any child of v whose parents are all scheduled, and put it into S with proper ordering.

7. Go to step 3.

This algorithm is a polynomial algorithm with $O(n^2)$, where n is the number of nodes in the DFG.

Branch and bound algorithm A schedule is an *active schedule* if we cannot find another schedule by simply shifting a scheduled node to an earlier starting time. It has been shown that an optimum schedule is an active schedule [3]. This algorithm exhaustively enumerates all *active schedules* to find the optimum schedule, and the branch and bound technique is used to reduce the search space. We will not discuss the detail of this algorithm, which can be found in related literature [3].

7 Examples

Two examples are presented in this section. The first design is a 16-bit unsigned add-and-shift multiplier, and the second design is a 16-point, 16-bit FIR digital filter. We will present these examples following the design procedure shown in Figure 1.

7.1 Multiplier

An asynchronous 16-bit unsigned add-and-shift multiplier takes an input, which is a ⟨16b, 16b⟩ multiplier-multiplicand pair, and produces an output, which is a 32*b* multiplication result.

DFG description Figure 39 is the input DFG description of the multiplier, where ASH, $f1$, and $f2$ are atomic functions. ASH (add-and-shift) takes the multiplicand (M), a partial multiplication result (A, Q'), and a partial multiplier (Q'') to generate a new partial multiplication result and a new partial multiplier by means of the following operations [12]:

$$\{ov, A\} \leftarrow A + M * Q_{LSB},$$

$$\{A, Q\} \leftarrow \{ov, A, Q_{MSB..LSB^{+1}}\},$$

where A, Q, and M are 16-bit, Q is formed by $\{Q', Q''\}$, ov is an 1-bit overflow for addition, and LSB^{+1} is the second least significant bit. $f1$ is an atomic function that assigns zero as the initial partial multiplication result into the input multiplier-multiplicand pair, i.e., $f1$: ⟨multiplier, multiplicand⟩ → ⟨16'h0000, multiplier, multiplicand⟩; $f2$ is the router R([1 o 1, 1 o 2, ..., 1 o 16, 2 o 1, 2 o 2, ..., 2 o 16]) that extracts the output of the multiplier from the output of the last ASH, i.e., $f2$: ⟨MS16b_product, LS16b_product, multiplicand⟩ → 32b_product.

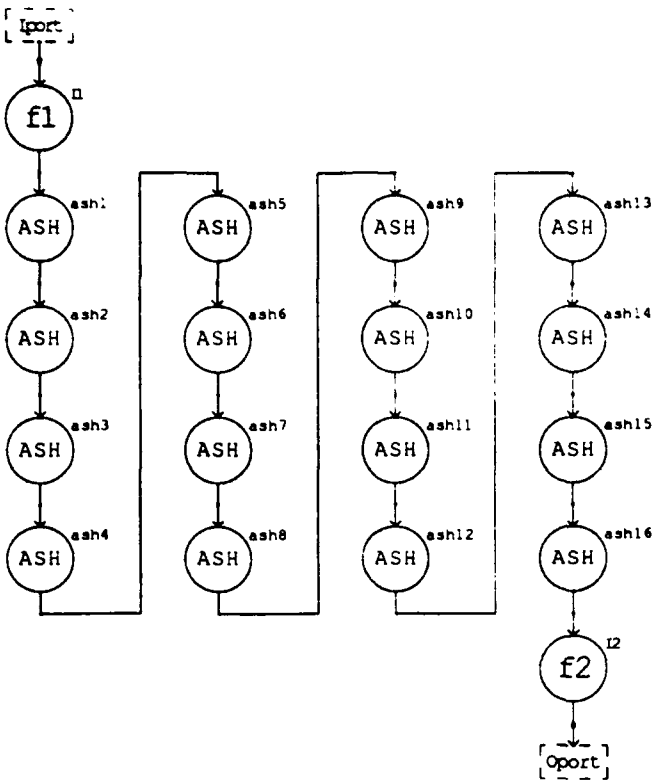


Figure 39: Input DFG description for the 16-bit add-and-shift multiplier.

Sequencing and allocation This example has a special structure with all atomic functions linearly connected. Initially we are more interested in optimizing the system throughput rate for this design, i.e., minimizing the pipeline period. Since we don't have an algorithm to solve this problem, we manually synthesized the design with one, two, four, and eight ASH units. Figure 40 shows the Gantt chart of the pipelined system synthesis result of the design with two ASH units, where the sixteen ASH operations be labeled add1 to add16 from system input to system output, and the two ASH units be labeled ASHER1 and ASHER2. We also synthesize this design by minimizing the system completion time. Figure 41 shows the Gantt chart of the non-pipelined system synthesis result of the design with two ASH units. Comparing Figure 40 and 41, we can easily observe different results for different objectives. We also find two ASH units are enough to have the minimum completion time for this design, i.e., more ASH units will not provide any better result. In the following steps, we only show the intermediate formats of our design procedure for the sequencing and allocation results of Figures 40 and 41.

Sharing schemes and local transformations The next step is to apply the sequencing and allocation result to the input DFG using the sharing scheme with fixed allocation and fixed sequence. Figures 42 and 43 are the mapped DFGs corresponding to the synthesis results in Figures 40 and 41, where the number of paths among ASHERs in Figure 42 and

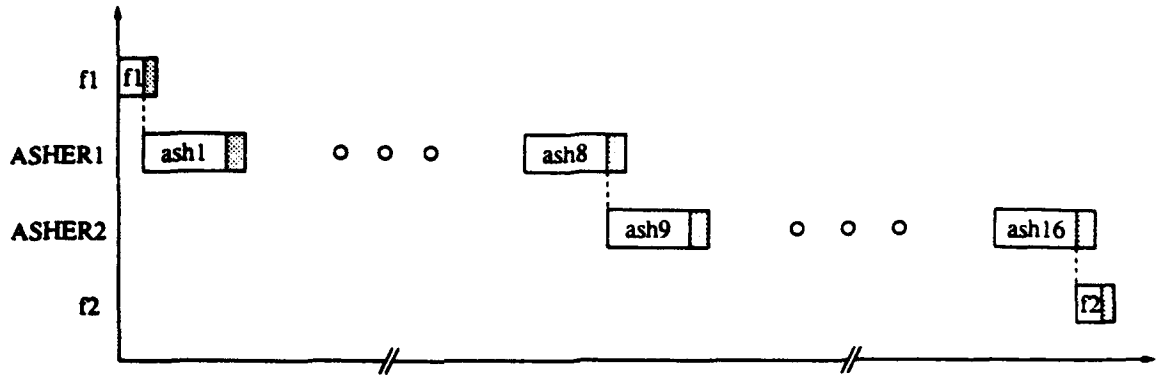


Figure 40: The Gantt chart for 2-ASH pipelined system synthesis.

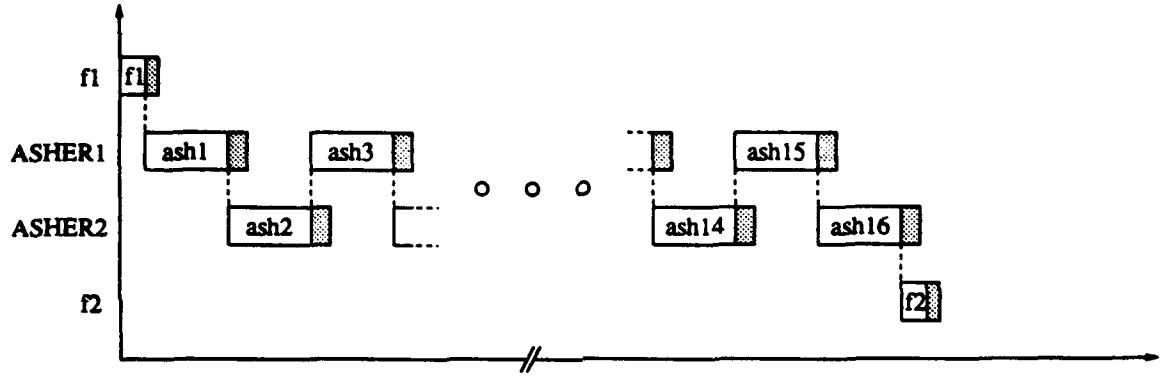


Figure 41: The Gantt chart for 2-ASH non-pipelined system synthesis.

the number of paths among ASHERs in Figure 43 are the same as the number in Figure 39, i.e., there is a separate path between each pair of add$i$ and add$(i+1)$ for $i = 1, \ldots 15$. After applying the general sharing scheme, we find that many paths can be merged. By applying the local transformation of Figure 37, Figure 42 is reduced to Figure 44, and Figure 43 is reduced to Figure 45. The control part of the sharing structure depends on how the inputs of the MSelector and the outputs of the MDistributors are routed. For example, in Figures 42 and 43, the control part CFfs8 generates the sequence (0,1,2,3,4,5,6,7) repeatedly for both the MSelector and the MDistributor, and the control part CFfs8' generates the sequence (0,1,2,3,4,5,6,7) repeatedly for the MSelector and the sequence (1,2,3,4,5,6,7,0) repeatedly for the MDistributor; in Figures 44 and 45, the control part CFfs8" generates the sequence (0,1,1,1,1,1,1,1) repeatedly for the MSelector and the sequence (1,1,1,1,1,1,1,0) repeatedly for the MDistributor.

Register minimization in EDFG Before the RTL netlist of the design is generated, we need to transform the DFG description into an EDFG description. We also can remove

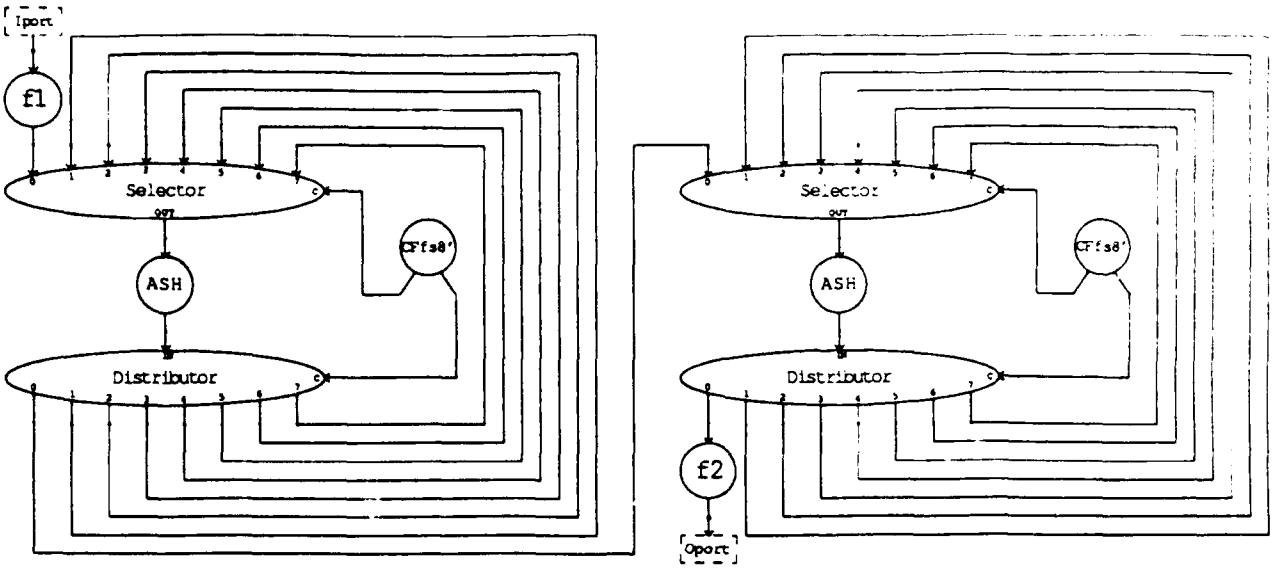


Figure 42: Mapped DFG description with two ASHERs for the pipelined system.

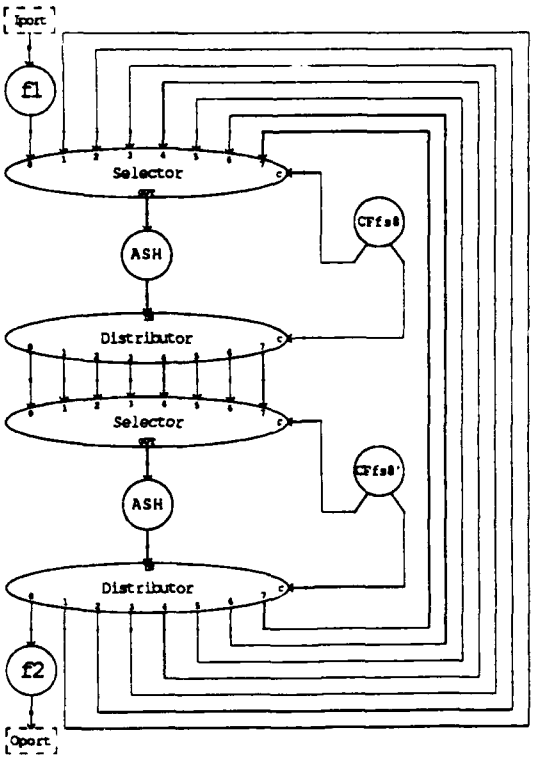


Figure 43: Mapped DFG description with two ASHERs for the non-pipelined system.

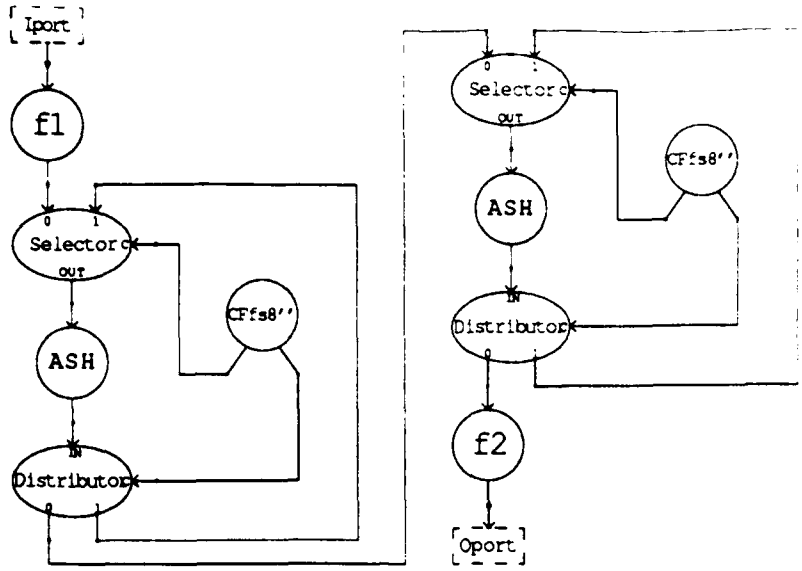


Figure 44: Reduced DFG description with two ASHERs for the pipelined system.

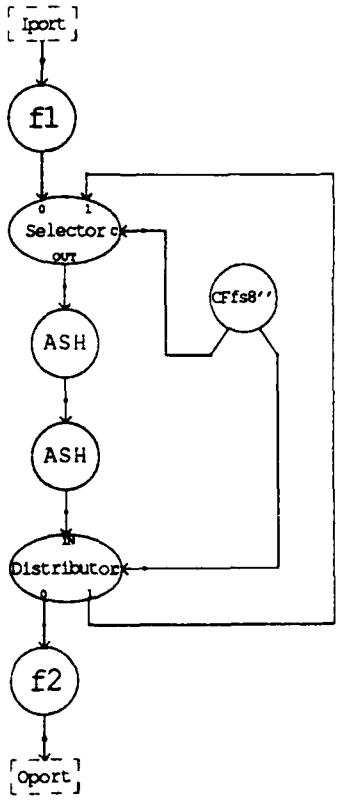


Figure 45: Reduced DFG description with two ASHERs for the non-pipelined system.

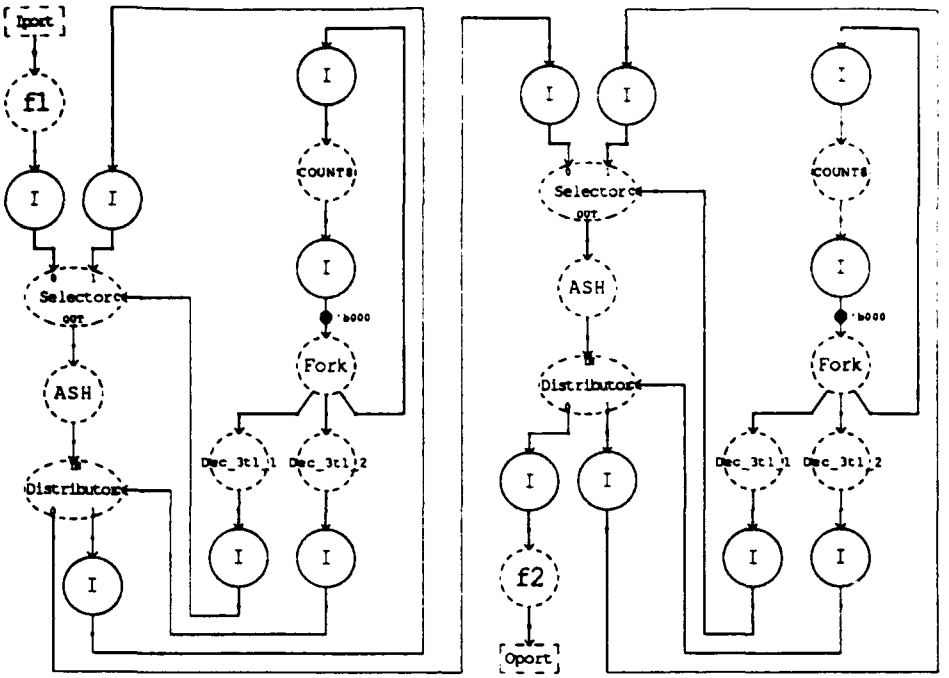


Figure 46: EDFG description with two ASHERs for the pipelined system.

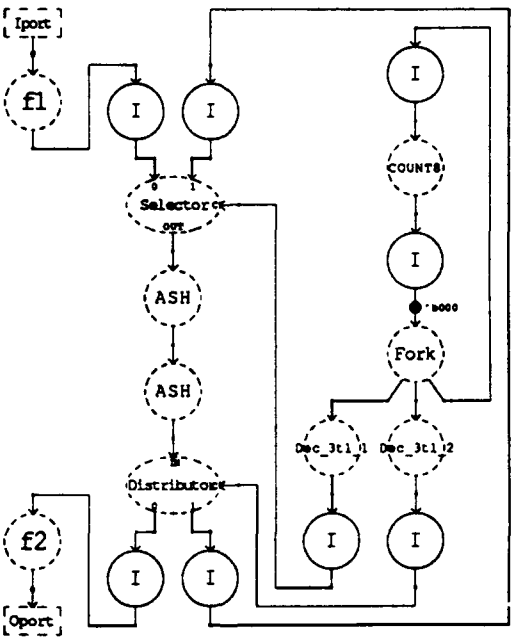


Figure 47: EDFG description with two ASHERs for the non-pipelined system.

unnecessary registers from the mapped EDFG description. Currently, we do not have an algorithm to deal with the register minimization problem, so we manually do the job in this design. Figures 46 and 47 are the EDFG descriptions corresponding to Figures 44 and 45 with some registers removed. In Figures 46 and 47, the loop of a COUNT8, two Is, and a 3-input phantom MFork forms a 3-bit counter, and Dec_2t1 and Dec_2t2 convert the 3-bit sequence (0,1,2,3,4,5,6,7) into the 1-bit sequence (0,1,1,1,1,1,1,1) and the 1-bit sequence (1,1,1,1,1,1,1,0), respectively.

The last step is to map the EDFG description into an RTL netlist for layout generation. The implementation results will be shown in Section 7.3.

7.2 FIR digital filter

The second design is a 16-point, 16-bit FIR digital filter. The convolution sum of the filter is

$$y(n) = \sum_{k=0}^{15} h(k) * x(n-k)$$

A causal FIR system with linear phase has the property that

$$h(k) = h(15-k), \quad \text{for } k = 0, \ldots, 7$$

Therefore, the convolution sum can be reduced to the following form,

$$\begin{aligned} y(n) &= \sum_{k=0}^{7} h(k) * (x(n-k) + x(n-15+k)) \\ &= h(0) * (x(n) + x(n-15)) + \\ &\quad h(1) * (x(n-1) + x(n-14)) + \\ &\quad \vdots \\ &\quad h(7) * (x(n-7) + x(n-8)) \end{aligned}$$

DFG description Figure 48 is the input DFG description of the FIR filter. There are two inputs for this system: in_H is the 8b input for the initialization of system coefficient $h(i)$ for $0 \leq i \leq 7$; in_X is the 16b input for $x(n)$ for $n \geq 0$. There is one 16b output, out_Y, for $y(n)$ for $n \geq 0$ in the system. In Figure 48 ADD, MUL, and $f3$ are atomic functions. We use a fixed-point computation to implement this design, where the 16-bit adder, ADD, and the 8-bit multiplier, MUL, are used to manipulate 16-bit data. $f3$ is the router $R([1,2,3,4,5,6,7,8])$, which truncates the last 8 bits from the 16-bit output of the ADD, so the following MUL can have a proper 8-bit input. In Figure 48, CFfs8.1 and mem_H are macro functions: CFfs8.1 generates the sequence (0,1,2,3,4,5,6,7) once after the system is started, so eight data tokens read in from in_H are distributed to proper mem_Hs for $h(0)$ to $h(7)$; mem_H, whose DFG description is shown in Figure 49, reads in a data

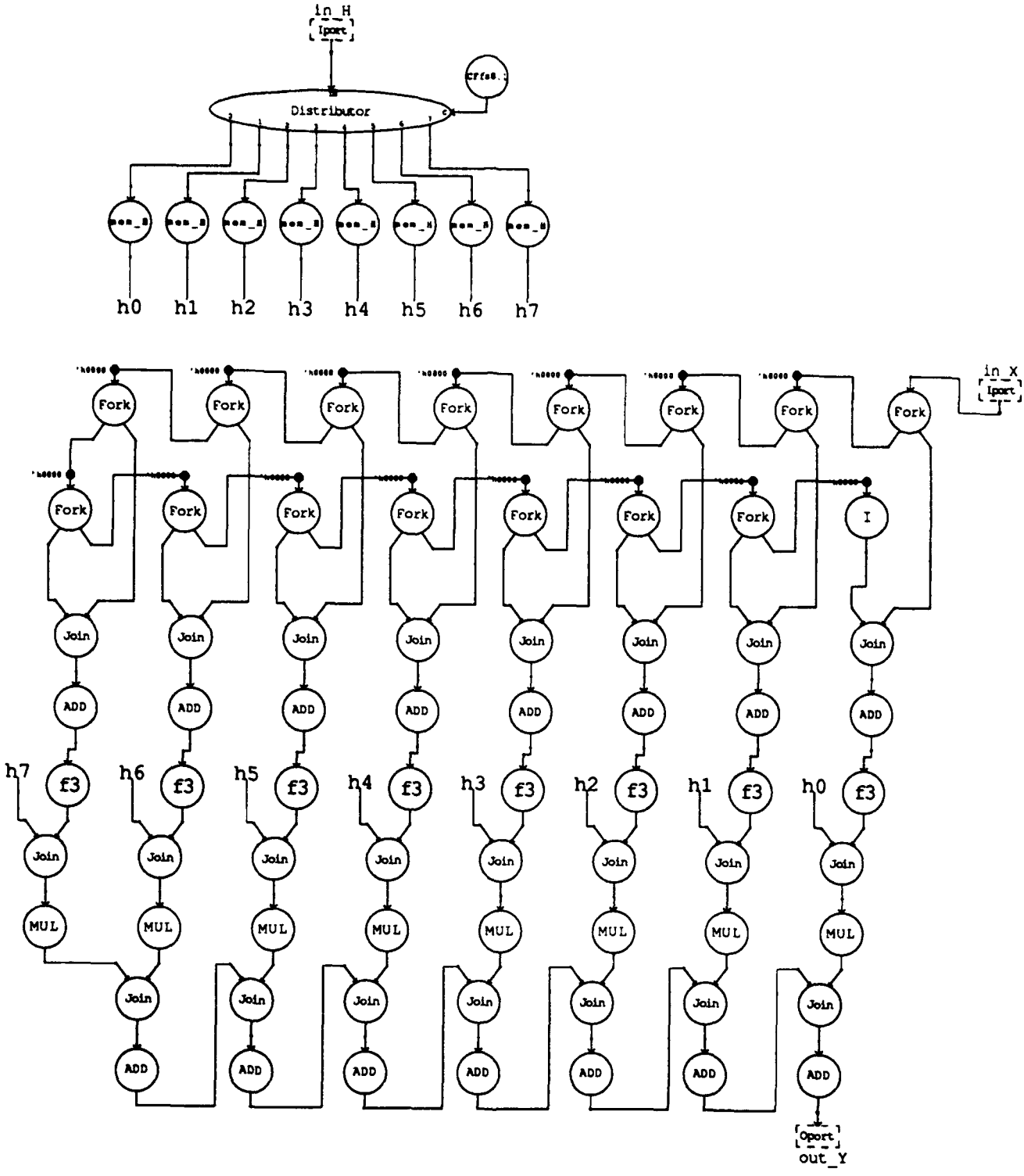
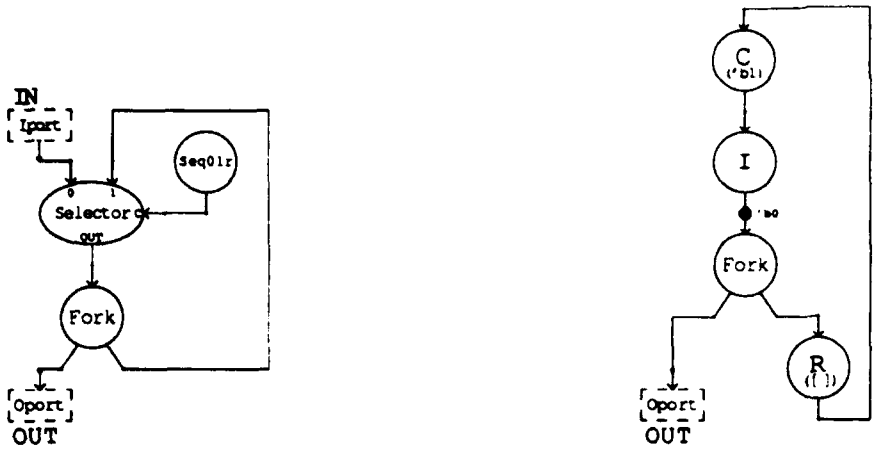


Figure 48: Input DFG description for the FIR digital filter.



(a) DFG description of mem_H (b) DFG description of Seq01r

Figure 49: DFG description for mem_H.

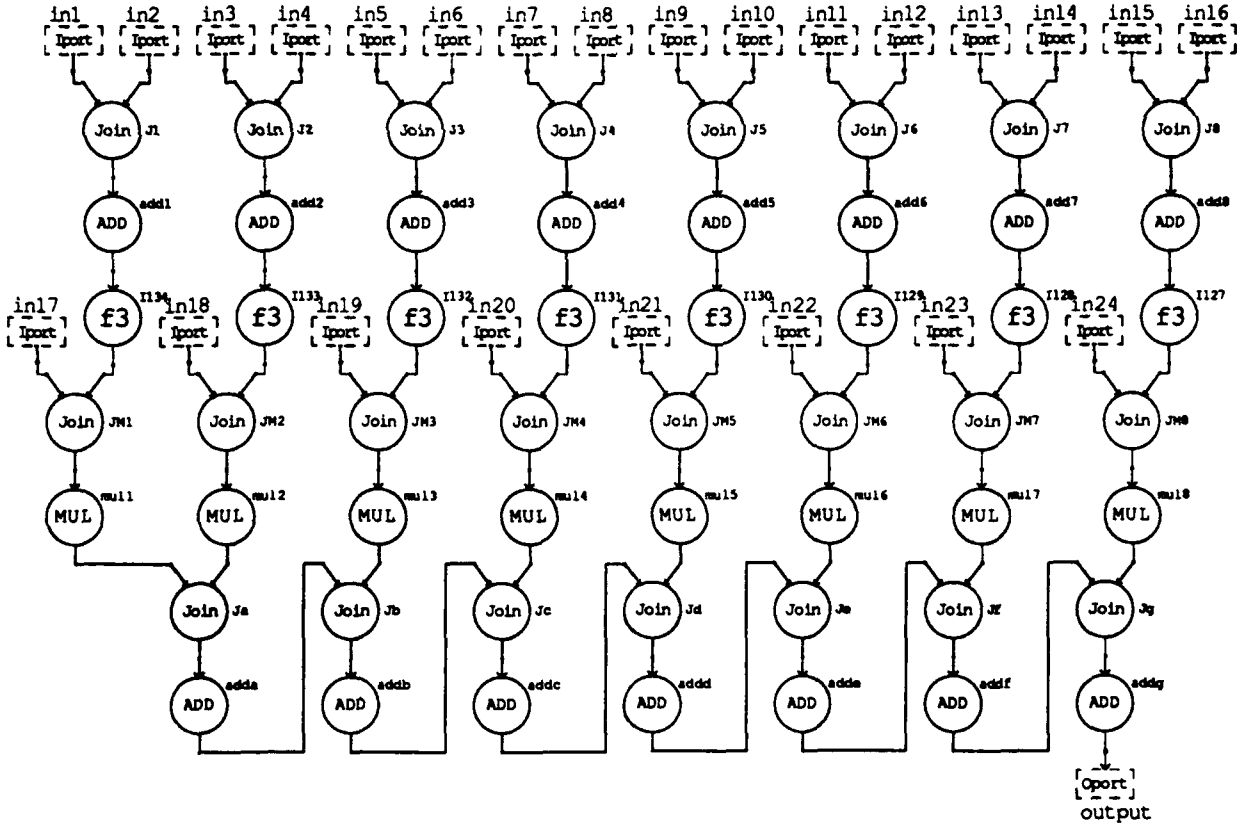


Figure 50: DFG description of the FIR filter for synthesis.

token from its input and keeps producing the same data token forever. There are fifteen data tokens in the DFG description of the FIR filter. These tokens represent the initial data of $x(n-k)$ for $1 \leq k \leq 15$, and they have the data value zero in this specification, i.e., $x(-1) = x(-2) = \ldots = x(-15) = 0$ with $n = 0$ initially. After $h(i)$ for $0 \leq i \leq 7$ are read, each input $x(n)$ will generate an output $y(n)$.

Construct Name	D_{FP} (nsec)	D_{BP} (nsec)
ADD	16.0	4.4
MUL	36.5	4.4
f3	0.0	0.0
MFork_2	0.0	3.1
MJoin_2	0.0	0.0
overhead	0.0	14.3

Table 1: Timing parameters for the FIR filter synthesis.

Resource Constraints (No. of operators)		Completion time (nsec)	
No. of MUL	No. of ADD	heuristic	Optimum
≥ 1	1	501.8	501.8
1	≥ 2	454.9	454.9
≥ 2	2	258.9	258.9
2	≥ 3	250.1	250.1
3	≥ 3	194.9	194.9
≥ 4	3	186.8	186.8
4	≥ 4	171.7	171.7
≥ 5	4	167.2	167.2
≥ 5	≥ 5	164.5	164.5

Table 2: The sequencing and allocation results of the 16-point FIR filter.

Sequencing and allocation There are three types of atomic functions in this design, but we only need to consider two of them, ADD and MUL, since the third, f3, can be implemented by physically truncating unused data. Currently we use one kind of implementation for each type of operation. Timing parameters for all implementations are shown in Table 1. D_{FP} of MFork_2 and D_{FP} labeled "overhead", which are both asterisked in Table 1, are zero due to the parallel of data computation and control generation in the hardware implementation. The overhead of the backward propagation delay time for the sharing scheme is 14.3 nsec. For convenience, we remove the token input part of the DFG, and Figure 50 shows the DFG without this token input part. We assume that all $x(n-i)$ for $i = 0, \ldots, 15$ and all $h(j)$ for $j = 0, \ldots, 7$ arrive at the same time, and we want to find the sequence and allocation with the

minimum system completion time for a given set of resources. Table 2 shows the sequencing and allocation results, where our heuristic algorithm always found the optimum solution for this example. In the following steps, we only show the intermediate format of our design procedure for a design with 2 MULs and 2 ADDs. The sequencing and allocation result of the 2-MUL, 2-ADD design is as follows, where the three columns give the computation starting time, the completion time, and the operator released time.

```
ADD1:
     add2: (  0.0   16.0   34.7)
     add3: ( 34.7   50.7   69.4)
     add5: ( 69.4   85.4  104.1)
     add6: (104.1  120.1  138.8)
     add7: (138.8  154.8  173.5)
     add8: (173.5  189.5  208.2)
     adde: (208.2  224.2  242.9)
     addg: (242.9  258.9  277.6)
ADD2:
     add1: (  0.0   16.0   34.7)
     add4: ( 34.7   50.7   69.4)
     adda: ( 69.4   85.4  104.1)
     addb: (107.7  123.7  142.4)
     addc: (142.4  158.4  177.1)
     addd: (177.1  193.1  211.8)
     addf: (224.2  240.2  258.9)
MUL1:
     mul2: ( 16.0   52.5   71.2)
     mul3: ( 71.2  107.7  126.4)
     mul5: (126.4  162.9  181.6)
     mul7: (181.6  218.1  236.8)
MUL2:
     mul1: ( 16.0   52.5   71.2)
     mul4: ( 71.2  107.7  126.4)
     mul6: (126.4  162.9  181.6)
     mul8: (189.5  226.0  244.7)
```

The corresponding Gantt chart of above result is shown in Figure 51.

Sharing schemes and local transformations The next step is to apply the sequencing and allocation result to the input DFG using the sharing scheme with fixed allocation and fixed sequence. Figure 52 is the mapped DFG corresponding to the synthesis results in Figure 51, where net labeling is used to represent data path connectivity. For example, add1 is connected to the output of J1 and to the right input of JM1 in Figure 50, and these two paths are labeled J1o and JM1r in Figure 52. Referring to Figure 51, add1 is scheduled as the first operation at operator ADD2, so J1o and JM1r are linked to the corresponding paths of the sharing structure for the ADD2 in Figure 52, i.e., input port 0 of MSelector and output port 0 of MDistributor for the ADD2. After applying the sharing scheme for the sequencing and allocation result, we are looking for possible reduction on the mapped DFG. In this design, there are two operands for each operation. In order to apply the same principle of the local transformation in Figure 37, we need to have corresponding inputs and outputs of

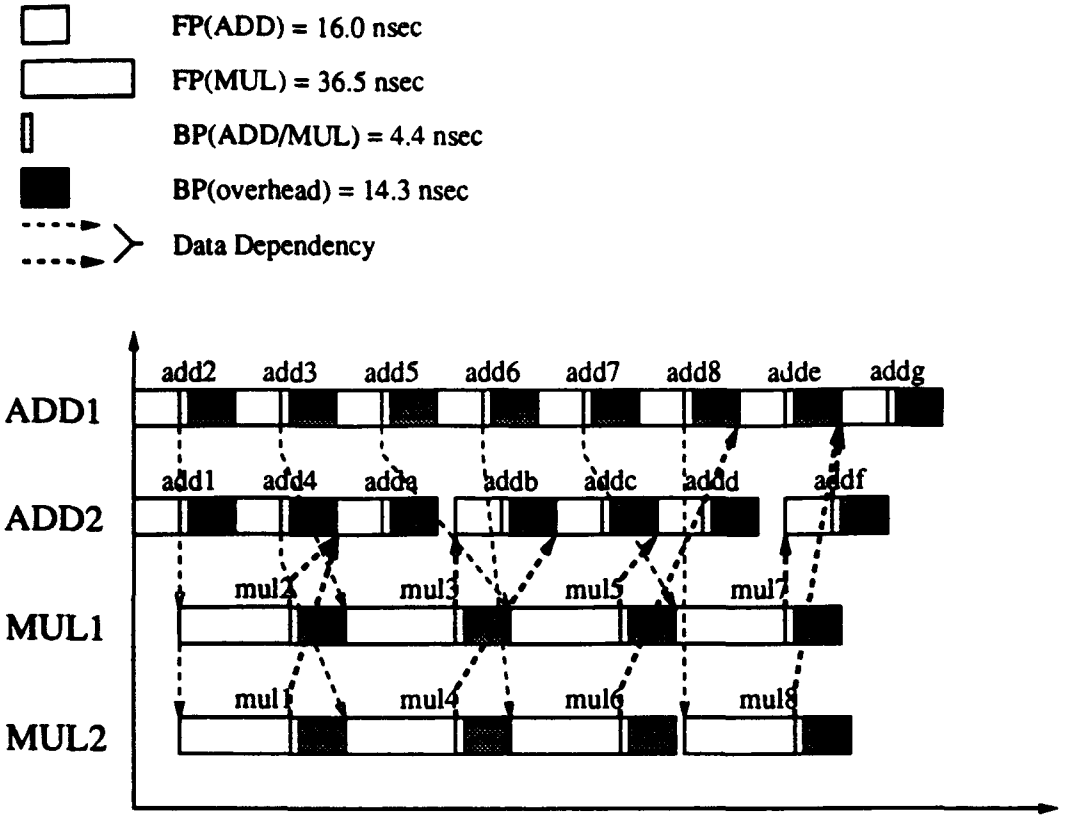


Figure 51: The Gantt chart for the 2-MUL, 2-ADD FIR filter.

two operations from the same sources and destinations. For example, ⟨Jbl, Jbr⟩ for addb and ⟨Jdl, Jdr⟩ for addd come from the same sources MDistributor of MUL1 for both left operands and MDistributor of ADD2 for both right operands, and they have the same destination, MSelector of ADD2, through Jbo and Jdo in Figure 52. Therefore, we merge these two sets to ⟨Jbdl, Jbdr⟩ and Jbdo. Similarly, ⟨Jel, Jer⟩ and Jeo for adde and ⟨Jgl, Jgr⟩ and Jgo for addg can be merged to ⟨Jegl, Jegr⟩ and Jego. Figure 52 is thus reduced to Figure 53. The control part of the sharing structure depends on how the inputs of MSelector and the outputs of MDistributors are routed. For example, in Figures 52 the control part CFfs4, CFfs7, and CFfs8 generate the sequence (0,1,2,3), the sequence (0,1,2,3,4,5,6), and the sequence (0,1,2,3,4,5,6,7) repeatedly for both MSelectors and MDistributors respectively. In Figure 53, after local transformations, the control part CFfs4' generates the sequence (0,1,2,3) repeatedly for the MSelector and the sequence (0,1,1,2) repeatedly for the MDistributor; the control part CFfs4" generates the sequence (0,1,2,3) repeatedly for the MSelector and the sequence (0,1,2,2) repeatedly for the MDistributor. Similarly, we can find the corresponding sequences generated by CFfs8' and CFfs7'.

Register minimization in EDFG After applying the sequencing and allocation result in Figure 53 to the original DFG description in Figure 48, we transformed the sequenced

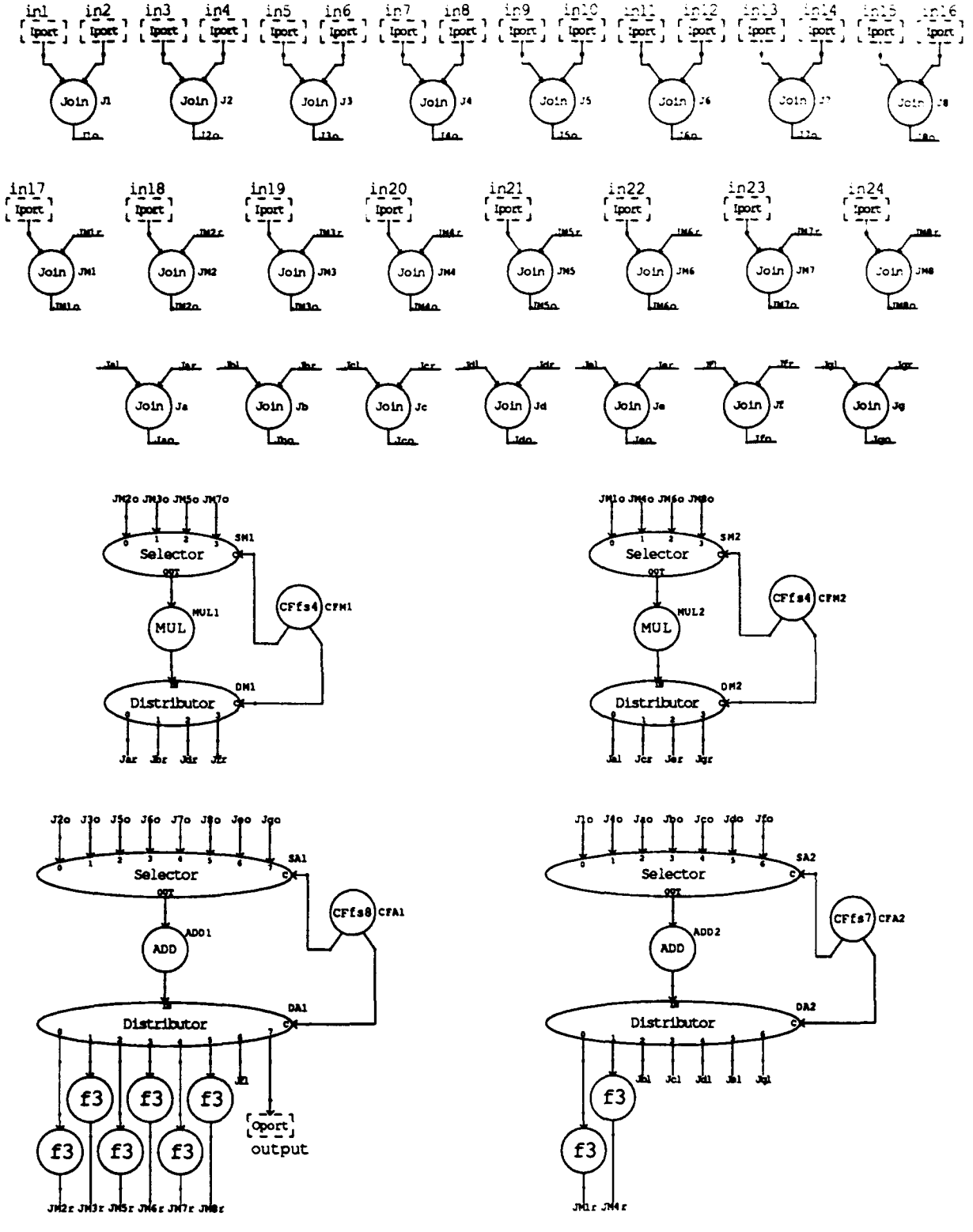


Figure 52: Mapped DFG description for the 2-MUL 2-ADD FIR design.

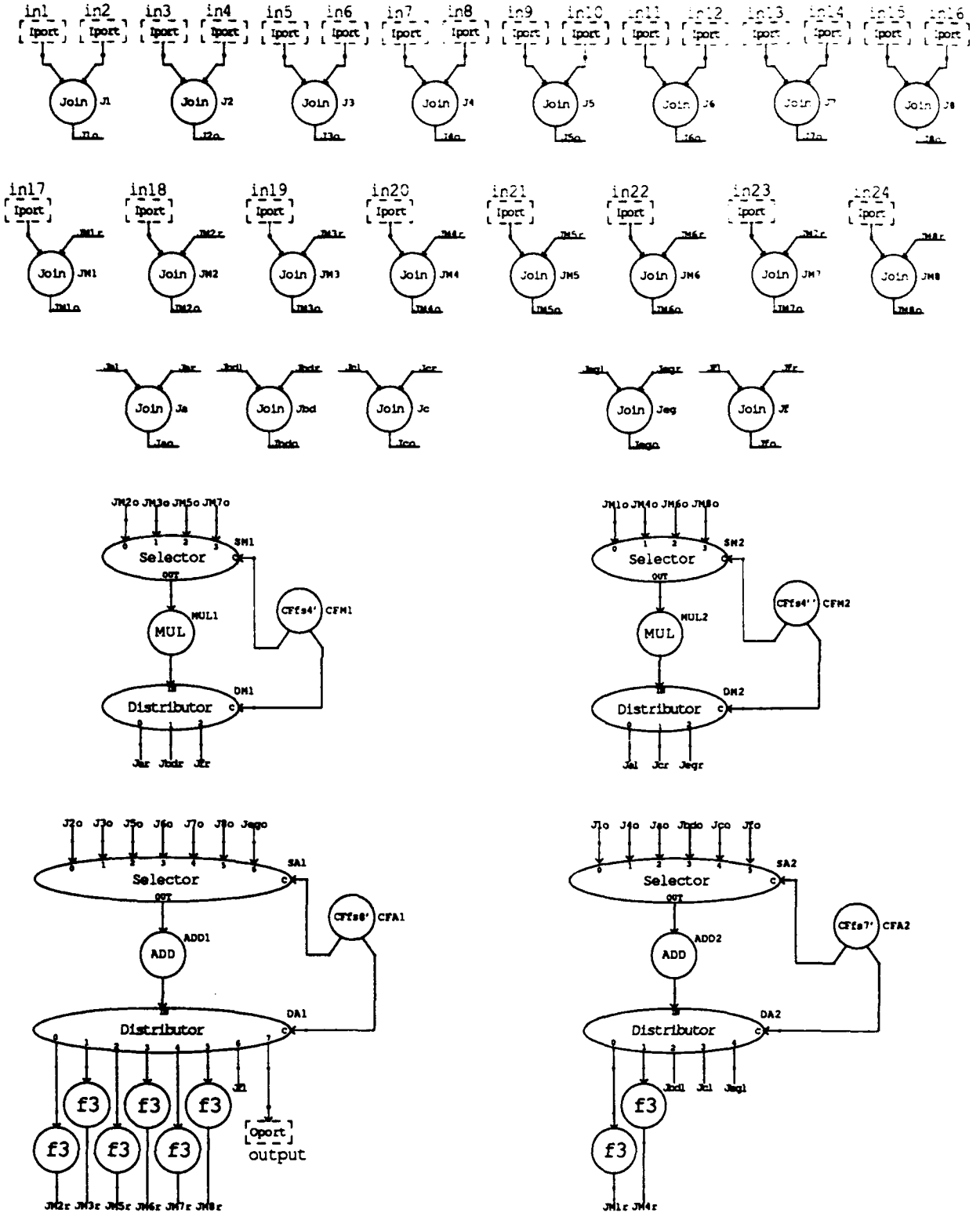


Figure 53: Reduced DFG description for the 2-MUL 2-ADD FIR design.

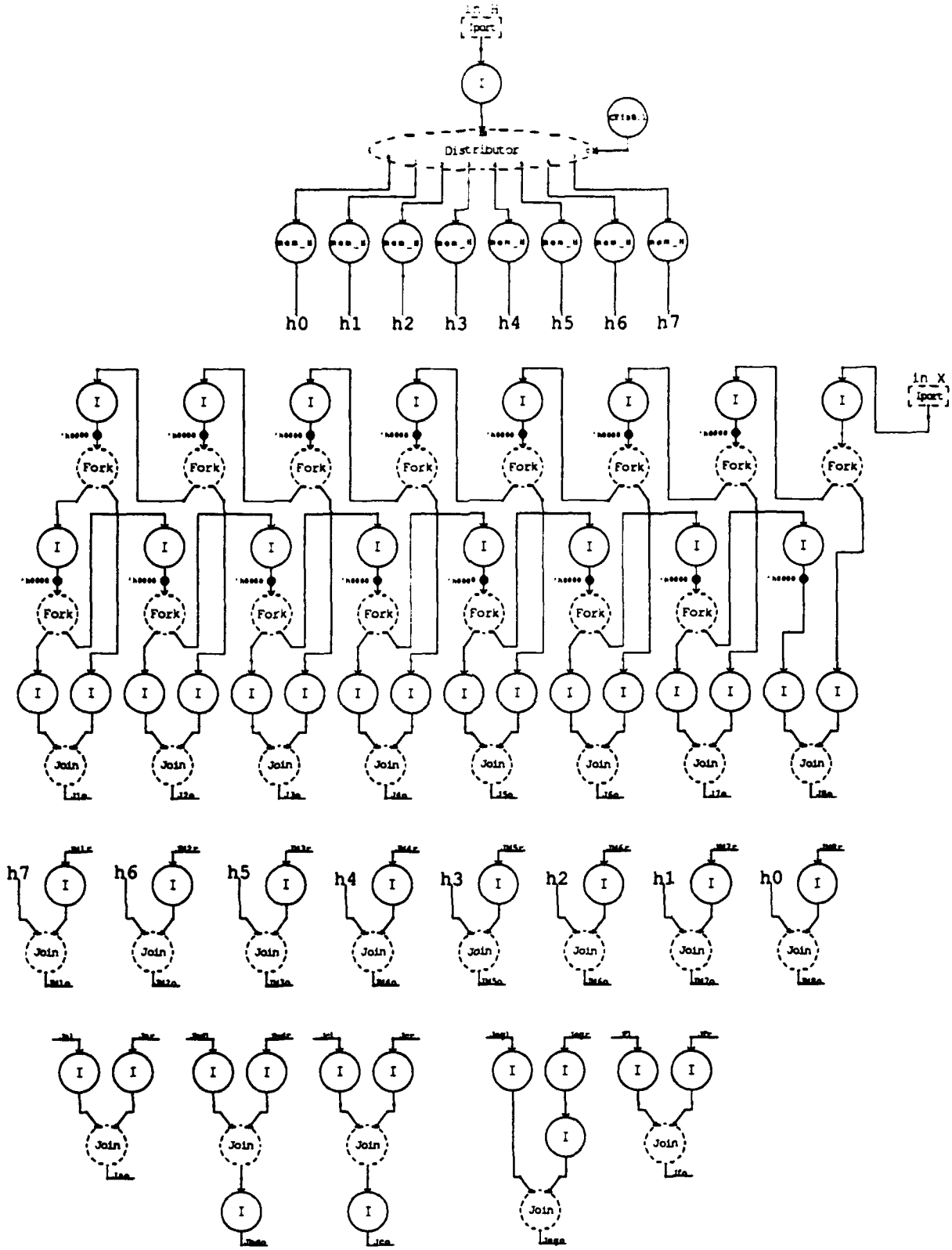


Figure 54: EDFG description for the 2-MUL 2-ADD FIR design (Part I).

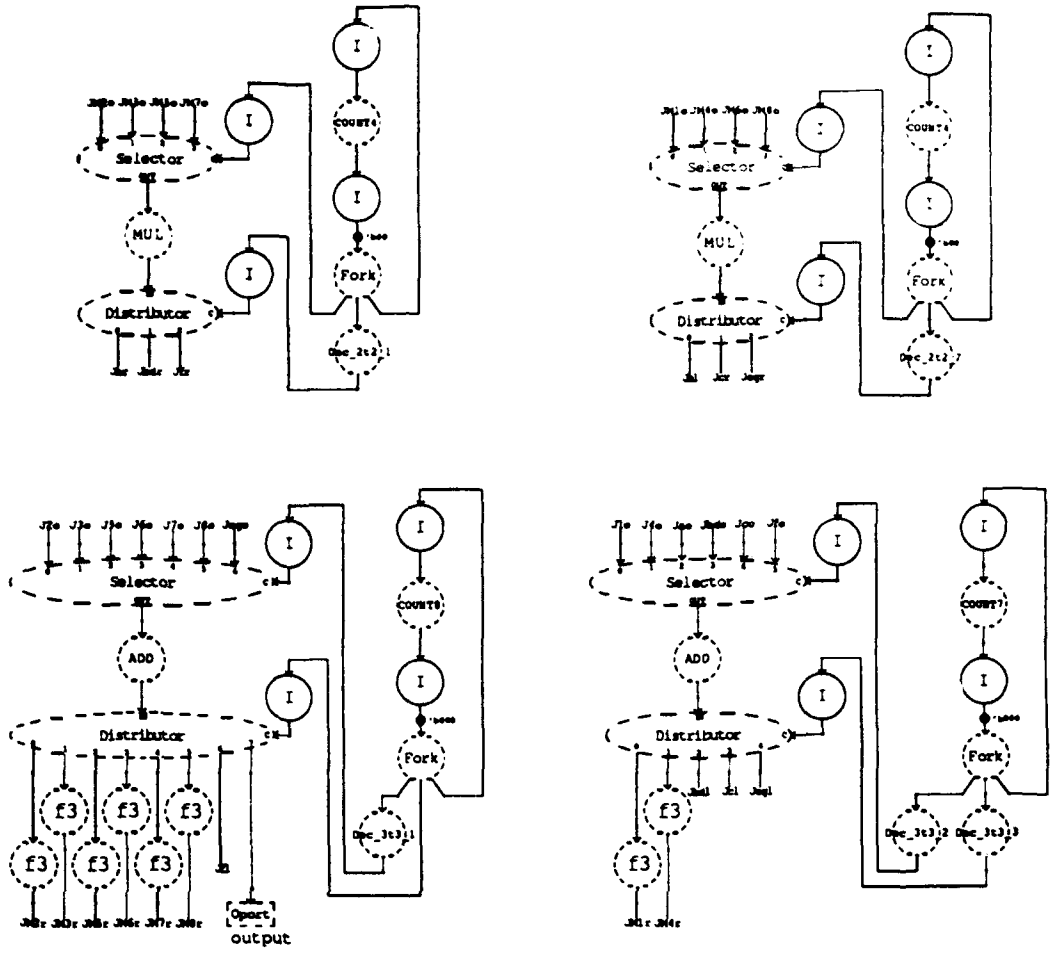
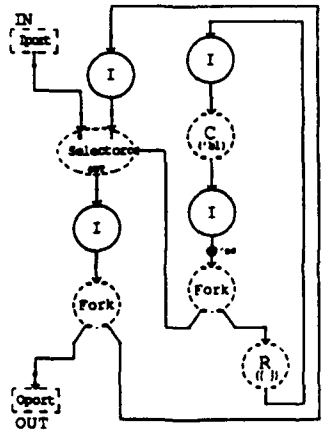
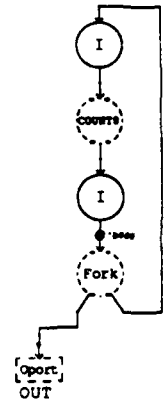


Figure 55: EDFG description for the 2-MUL 2-ADD FIR design (Part II).





(a) EDFG description of mem_H (b) EDFG description of CFfs8.1

Figure 56: EDFG description for mem_H and CFfs8.1.

DFG description into EDFG description. Then we removed unnecessary registers from the mapped EDFG description. Figures 54 and 55 are the final EDFG description after manual register minimization, where the EDFG descriptions of mem_H and CFfs8.1 are shown in Figure 56. In the final EDFG description, COUNT4, COUNT7 and COUNT8 are used to produce the sequences (0,1,2,3), (0,1,2,...,6), and (0,1,2,...,7) respectively, and Dec_$it j_k$ is a decoder function which converts a i-bit control signal to j-bit control signal with index k to distinct different decoders.

The last step is to map the EDFG description into RTL netlist for the layout generation. The implementation results will be shown in Section 7.3.

7.3 Experimental Results

We have implemented the example designs discussed in the preceding section using a library of asynchronous building blocks [32] composed with an industrial standard cell library, HP C34100 [34], in a commercial CAD tool, Cadence Design Framework II^{TM}. Both the RTL netlist and the layout of these designs were produced [10]. The performance of each design has been simulated in a mix-mode simulator, Verilog-XL$^{(R)}$, using the model distributed with the cell library, plus extracted wiring capacitances. The implementation and the simulation of these designs at the layout level show the feasibility of our design method. To show the effectiveness of the data flow model, we further compare the area/performance of these designs obtained from our DFG/EDFG model with the area/performance obtained from the final layouts.

To use the DFG/EDFG model, we need to know all the timing parameters, such as D_{sfl}, D_{sbl}, D_{sp}, D_{fp}, and D_{bp}, during the course of synthesis procedure. Since the asynchronous building blocks have been designed, these timing parameters are obtained by simulation. Table 3 and Table 5 are the EDFG timing parameters for the multiplier design and for the FIR design, respectively. The fanout/loading capacitance internal to each block is considered, but the fanout/loading capacitance external to the block, which depends on the interconnection of the block in a real design, is not considered currently. In Table 5, several modules, e.g., MSelector_4, have more than one set of timing parameters, which represent more than one implementation for the same EDFG construct. The slow module is used for non-critical nodes of the FIR filter design. We use "DFGsim" to label the performance measure of the design from the simulation of the DFG/EDFG model. At the DFG/EDFG level, the area measure of a design is the area sum of all asynchronous blocks used in the design, and the area of each block is the area sum of all standard cells which implement the block. Therefore, no wiring area is considered at the DFG/EDFG level. We use "Cell" to label the area measure.

The performance for a real layout is obtained by the simulation of standard cell netlist

[10]Design Framework II is a design framework, and it is a trademark of Cadence Design Systems, Inc. Cell Ensemble is a standard cell placement and routing tool used in our experiments for the layout generation, and it is a trademark of Cadence Design Systems, Inc. Verilog-XL is a mix-mode simulator, and it is a registered trademarks of Cadence Design Systems, Inc. DRACULA is an IC layout verification system, and it is a registered trademark of Cadence Design Systems, Inc.

Construct Name	D_{fp} (nsec)	D_{bp} (nsec)	D_{sfl} (nsec)	D_{sbl} (nsec)	D_{sp} (nsce)
(Ph) f1	0.00	0.00	—	—	—
(Ph) f2	0.00	0.00	—	—	—
(Ph) ASH	14.70	0.00	—	—	—
(Ph) COUNTx	1.70	0.00	—	—	—
(Ph) Dec_itj_k	2.10	0.00	—	—	—
(Ph) MSelector_2	6.03	4.85	—	—	—
(Ph) MDistributor_2	9.19	1.53	—	—	—
(Ph) MFork_3	0.00	3.10	—	—	—
Storage($\langle$16b,16b,16b$\rangle$)	—	—	3.80	4.80	0
Storage(nb), for $n \leq 4$	—	—	2.73	3.77	0

Table 3: Timing parameters for the EDFG in the experiment of the multiplier design.

Number of shared units	Completion time (nsec)			Pipeline period (nsec)			Area (x10^6 μ^2)		
	Csim	DFGsim	Ratio	Csim	DFGsim	Ratio	Core	Cell	Ratio
16	311.14	299.80	0.961	24.25	23.30	0.964	12.377	7.236	0.585
8	418.66	397.56	0.950	65.38	60.40	0.924	16.302	9.857	0.605
4	475.61	442.28	0.930	131.51	120.80	0.919	8.532	5.183	0.607
2 (pipeline)	504.10	464.64	0.922	263.80	241.60	0.916	4.231	2.605	0.616
1	519.70	476.02	0.916	528.82	483.20	0.914	2.124	1.371	0.645
2 (non-pipe)	371.82	348.32	0.937	381.52	359.20	0.941	2.771	1.671	0.603

Table 4: Experimental results of the multiplier design.

with wiring capacitances derived from a parasitic extraction tool, DRACULA$^{(R)}$, and it is labeled "Csim". The area for a real layout is obtained by the multiplication of the width and the height of the layout. The area measured for the core size of the final layout is labeled "Core".

Table 4 gives the experimental results of the 16-bit multiplier. Figures 57 and 58 show two layouts for the multiplier implementation. We found that our performance estimation from the simulation of the DFG/EDFG model is within 91.6% to 96.4% of the final layout performance measurement. The experimental result also shows that the cell area, which is used as the area measurement in the DFG/EDFG model, approximately occupies 58.5% to 64.5% of the final layout.

Table 6 gives the experimental results of the 16-point 16-bit FIR filter. Figure 59 shows one layout for the FIR filter implementation. Again we found that the performance estimation from the simulation of the DFG/EDFG model is within 87.6% to 98.1% of the final layout performance measurement. This experimental result also shows that the cell area obtained from the DFG/EDFG model approximately occupies 56.2% to 59.4% of the final

Construct Name	D_{fp} (nsec)	D_{bp} (nsec)	D_{sfl} (nsec)	D_{sbl} (nsec)	D_{sp} (nsce)
(Ph) f3	0.00	0.00	—	—	—
(Ph) C('b1)	0.00	0.00	—	—	—
(Ph) ADD	11.60	0.00	—	—	—
(Ph) MUL	32.10	0.00	—	—	—
(Ph) COUNTx	1.70	0.00	—	—	—
(Ph) Dec_it$j$_k	2.10	0.00	—	—	—
(Ph) MSelector_2	8.87	6.28	—	—	—
(Ph) MSelector_3	8.92	5.91	—	—	—
(Ph) MSelector_4 #1	5.29	6.13	—	—	—
(Ph) MSelector_4 #2	8.92	6.59	—	—	—
(Ph) MSelector_5	6.77	6.90	—	—	—
(Ph) MSelector_6	7.08	6.50	—	—	—
(Ph) MSelector_7	6.87	6.72	—	—	—
(Ph) MSelector_8	10.26	7.64	—	—	—
(Ph) MSelector_10	6.51	7.77	—	—	—
(Ph) MDistributor_2	9.19	1.53	—	—	—
(Ph) MDistributor_3 #1	5.18	2.68	—	—	—
(Ph) MDistributor_3 #2	8.72	2.84	—	—	—
(Ph) MDistributor_5	5.96	3.72	—	—	—
(Ph) MDistributor_8 #1	6.30	3.75	—	—	—
(Ph) MDistributor_8 #2	11.00	3.92	—	—	—
(Ph) MDistributor_10	6.96	5.66	—	—	—
(Ph) MFork_2	0.00	3.10	—	—	—
(Ph) MFork_3	0.00	3.10	—	—	—
(Ph) MJoin_2	3.10	0.00	—	—	—
Storage(⟨8b,8b⟩)	—	—	2.80	3.83	0
Storage(⟨16b,16b⟩)	—	—	2.80	3.83	0
Storage(8b) #1	—	—	2.80	3.83	0
Storage(8b) #2	—	—	4.25	5.27	0
Storage(16b) #1	—	—	2.80	3.83	0
Storage(16b) #2	—	—	4.25	5.27	0
Storage(nb), for $n \leq 3$	—	—	2.73	3.77	0

Table 5: Timing parameters for the EDFG in the experiment of the FIR design.

# of shared units		Completion time (nsec)			Pipeline period (nsec)			Area (x10^6 μ^2)		
mult.	adder	Csim	DFGsim	Ratio	Csim	DFGsim	Ratio	Core	Cell	Ratio
1	1	615.84	539.40	0.876	627.20	552.41	0.881	16.434	9.583	0.583
2	2	298.53	273.85	0.917	307.32	283.90	0.924	20.923	12.419	0.594
3	3	210.40	202.68	0.963	217.01	212.88	0.981	23.707	13.318	0.562

Table 6: Experimental results of the FIR design.

layout.

The Csim (actual extracted) value is larger than the DFGsim (estimated) value for each design, due to the following reasons.

1. The operation fanout and the control fanout external to the basic blocks are not considered in the current model.

2. The wiring delays between modules are also not considered in the current model; this delay cannot be accurately estimated until the actual layout is generated.

3. Some extra buffers were needed between some modules to comply with a CAD tool limitation in the current implementation, and these extra delays were not known during synthesis and analysis of the design at the DFG/EDFG model.

Despite these factors, our high-level timing model is quite accurate; the DFGsim/Csim ratio is 98.1% for the best case and is 87.6% for the worst case in all our experimental results.

It is common to use the cell area to estimate the routing overhead before placement and routing [37]. From above two examples, the Cell/Core ratio is within 56.2% to 64.5%. Although the area ratios for the above design are not fixed, these ratios vary in a certain small range. Therefore, the area measure by the cell area in the data flow model is sufficient for high-level synthesis algorithms.

Since we have an accuracy timing model and a proper area measurement at the data flow level, synthesis results represent the design space properly. One of the main reasons that synthesis algorithms explore the design space properly is that in our system the area/performance overhead of resource sharing, such as control units and multiplexers/demultiplexers, are accurately reflected and predicted. This allows us to estimate both the performance and the area at the data flow level quite accurately.

8 Conclusion

In this report, we have presented a design method for asynchronous systems based on the data flow specification. An asynchronous system is seen as a set of communicating processes, and the token data flow model is used to describe the behavior of the system. This design method not only provides a high-level description language, but also provides a systematic transformation within the data flow model to support high-level synthesis such as sequencing and allocation, sharing schemes, local transformations, and register minimization. Finally, the data flow specification is transformed into an EDFG description which is sufficient for layout realization.

In order to make the synthesis result useful, we have derived a timing model for the data flow specification so that the synthesis algorithms have an accurate and practical model. Experimental results show the effectiveness of using the timed data flow specification to design, analyze and synthesize asynchronous systems. Experimental results also show that

the accuracy of our timing model at the data flow model is within 90% of the actual implementation.

Many other issues regarding the design of basic asynchronous blocks and the hardware implementation have not been covered in this report, but they play an important role in realizing and demonstrating this design method. The detail of the design of basic asynchronous blocks can be found in [32].

Acknowledgement

The authors thank Cesar Pina and Wes Hansford for offering the circuit design environment at MOSIS, allowing us to easily verify our ideas. We appreciate the use of the MOSIS netlist-to-parts service, standard cell library, and implementation tools. The authors also thank Professor Alice Parker for her critiques and comments which have sharpened the this work in high-level synthesis. The authors also thank Jeff Sondeen for his encouragement and many helpful discussions. He not only corrected our writing errors but provided critiques and comments which have sharpened the ideas of this report. The authors also thank Anne Marie Edenhofner for her friendship and her patience to review the preliminary version of this report. She not only corrected several grammatical errors but also discussed with us the issue of writing style.

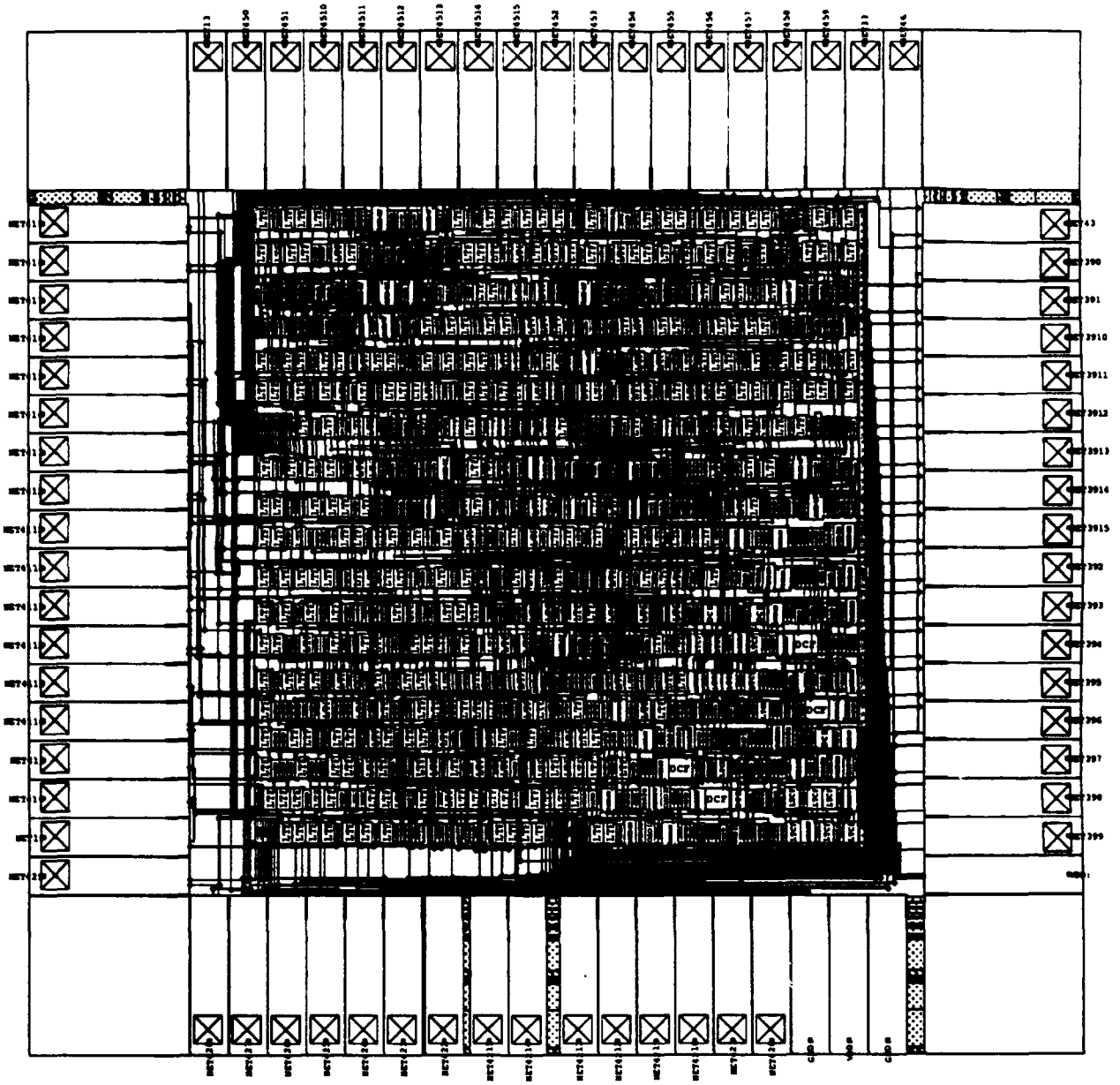


Figure 57: Layout for the 2-ASH pipelined multiplier design.

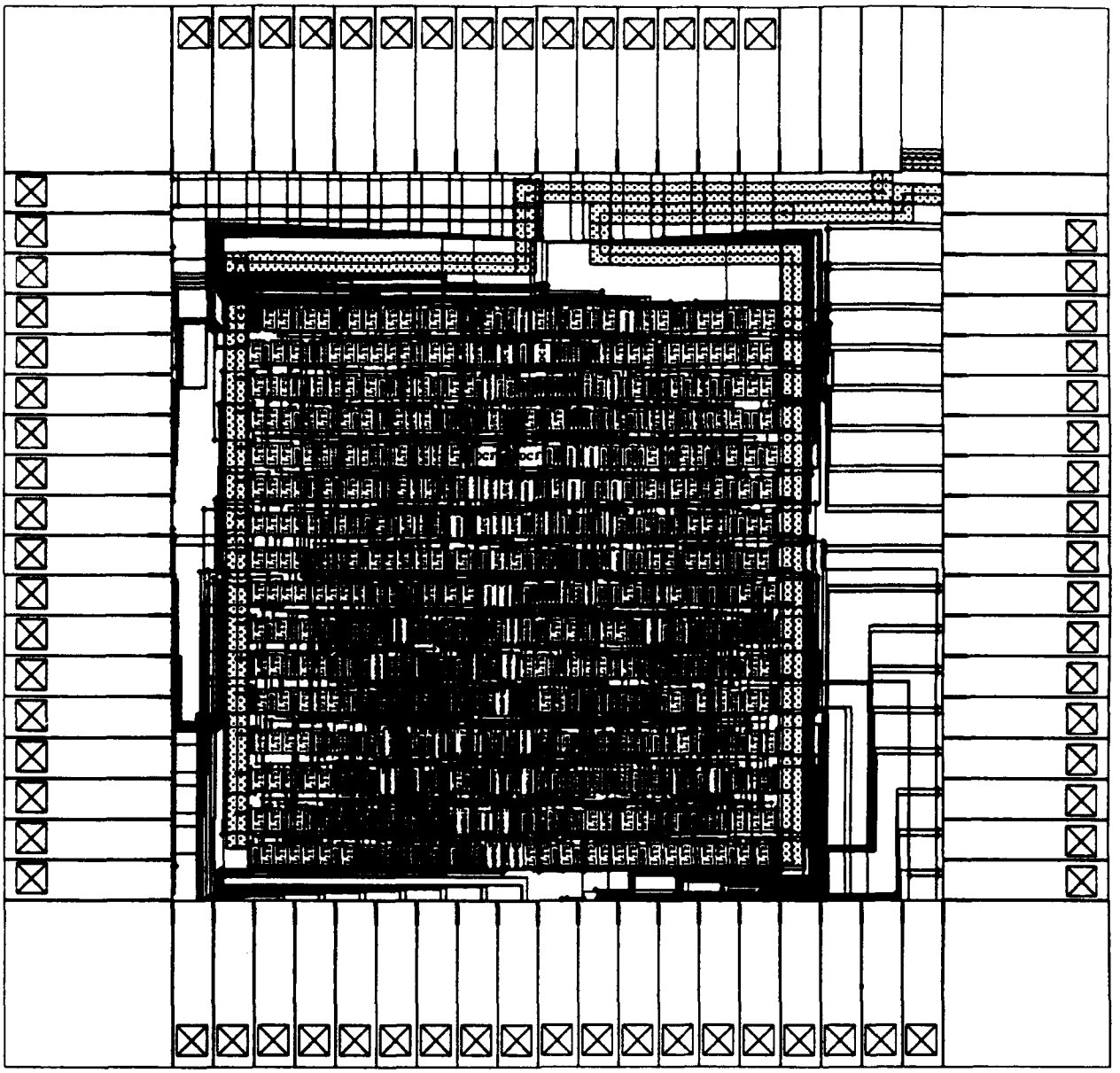


Figure 58: Layout for the 2-ASH non-pipelined multiplier design.

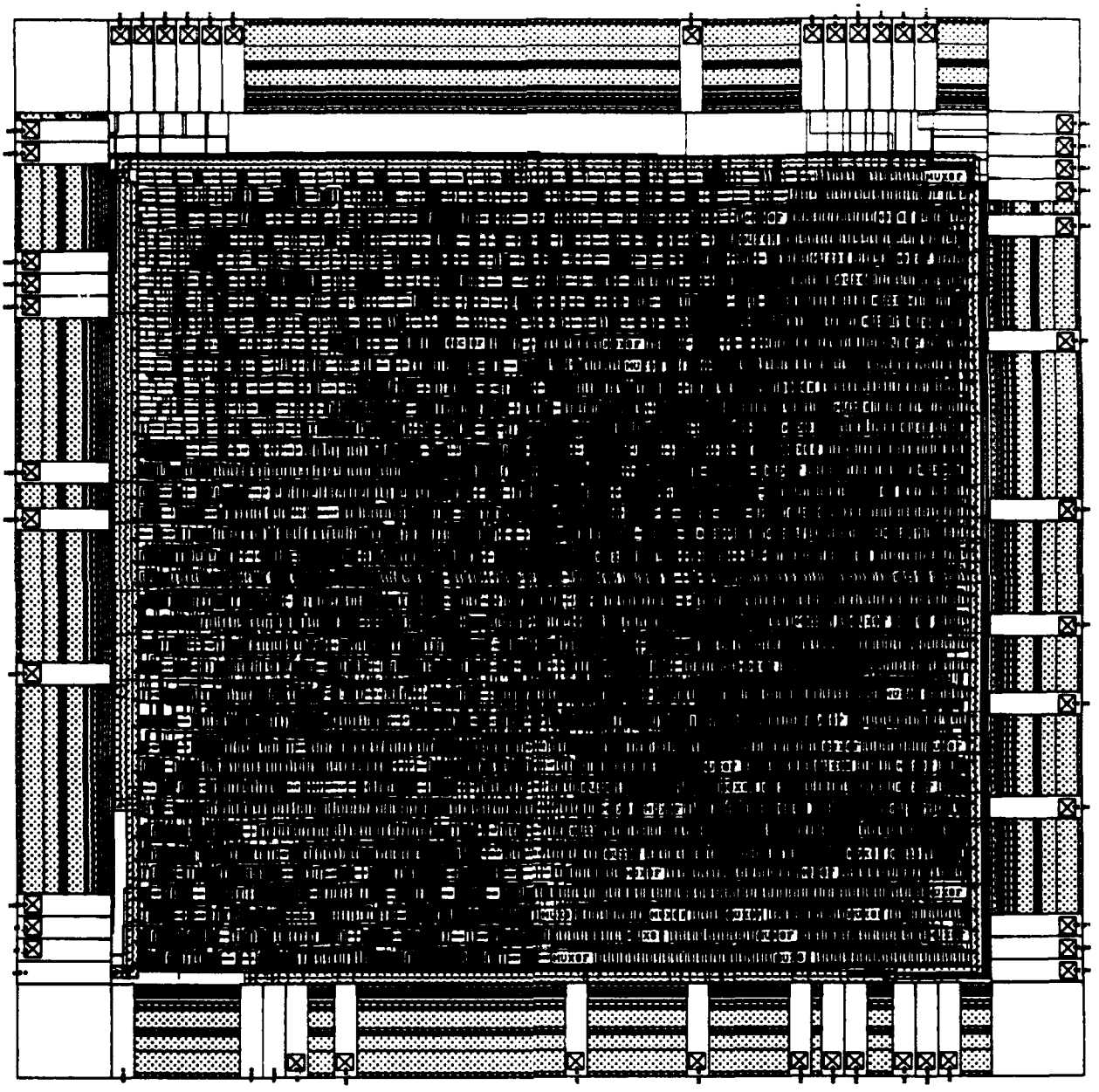


Figure 59: Layout for the 2-ADD 2-MUL FIR design.

References

[1] V. Akella and G. Gopalakrishnan. "SHILPA: A High-Level Synthesis System for Self-Timed Circuits". In *Proceedings of the ICCAD-92*, pages 587–591, 1992.

[2] J. Backus. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", The 1977 Turing Award Lecture. *Communications of the ACM*, 21(8):613–641, 1978.

[3] K. R. Baker. *Introduction to Sequencing and Scheduling.* John Wiley & Sons, 1974.

[4] E. Brunvand. Translating Concurrent Communicating Programs into Asynchronous Circuits. Technical Report CMU-CS-91-198, School of Computer Science, Carnegie Mellon University, September 1991.

[5] E. Brunvand and R. F. Sproull. Translating Concurrent Communicating Programs into Delay-Insensitive Circuits. Technical Report CMU-CS-89-126, School of Computer Science, Carnegie Mellon University, April 1989.

[6] S. M. Burns and A. J. Martin. Synthesis of Self-timed Circuits by Program Transformation. Technical Report 5253:TR:87, Dept. of Computer Science, California Institute of Technology, 1987.

[7] T.-A. Chu. Synthesis of Self-timed VLSI Circuits from Graph Theoretic Specifications. Technical Report PhD thesis, Dept. of EECS, Massachusetts Institute of Technology, September 1987.

[8] A. L. Davis and R. M. Keller. "Data Flow Program Graphs". *IEEE COMPUTER*, 15(2):26–41, 1982.

[9] J. C. Ebergen. Arbiters: An Exercise in Specifying and Decomposing Asynchronously Communicating Components. Technical Report Technique Report CS-90-29, Dept. of Computer Science, University of Waterloo, July 1990.

[10] A. D. Friedman and P. R. Menon. *Theory and Design of Switching Circuits.* Computer Science Press, Inc., 1975.

[11] D. Gajski et al. *High-Level Synthesis – Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[12] J. P. Hayes. *Computer Architecture and Organization.* McGraw-Hill, New York, second edition, 1988.

[13] I. Koren and G. M. Silberman. "A Direct Mapping of Algorithms onto VLSI Processing Arrays Based on the Data Flow Approach". In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 335–337, 1983.

[14] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. "Algorithms for Synthesis of Hazard-free Asynchronous Circuits". In *Proceedings of the 28th Design Automation Conference*, pages 302–308, 1991.

[15] K.-J. Lin and C.-S. Lin. "Automatic Synthesis of Asynchronous Circuits". In *Proceedings of the 28th Design Automation Conference*, pages 296–301, 1991.

[16] A. J. Martin et al. The Design of an Asynchronous Microprocessor. *Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373, March 1989.

[17] E. J. McCluskey. *Introduction to the Theory of Switching Circuits*. McGraw-Hill, New York, 1965.

[18] M. C. McFarland, A. C. Parker, and P. Camposano. "The High-Level Synthesis of Digital Systems". *Proceedings of the IEEE*, 78(2):301–318, 1990.

[19] B. Mendelson, B. Patel, and I. Koren. "designing special-purpose co-processors using the data-flow paradigm". In *Advanced Topics in Data-Flow Computing*. editted by J.-L. Gaudiot and L. Bic, Prentice-Hall Inc., pages 547–570, 1991.

[20] B. Mendelson and G. M. Silberman. "Mapping Data Flow Programs on a VLSI Array of Processors". In *Proceedings of International Symposium on Computer Architecture*, pages 72–80, 1987.

[21] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messerschmitt. "Automatic Synthesis of Asynchronous Circuits from High-Level Specifications". *IEEE Transactions on Computer-Aided Design*, 8(11):1185–1205, 1989.

[22] R. H. Mohring. "algorithmic aspects of comparability graphs and interval graphs". In *Graphs and Orders - The Role of Graphs in the Theory of Ordered Sets and Its Applications*. editted by I. Rival, NATO ASI Series, pages 41–101, 1984.

[23] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger. "Synthesis of Delay-Insensitive Modules". In *1985 Chapel Hill Conference on VLSI*, pages 67–86, 1985.

[24] N. Park and A. C. Parker. "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications". *IEEE Transactions on Computer-Aided Design*, 7(3):356–370, March 1988.

[25] C. V. Ramamoorthy and G. S. Ho. "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets. *IEEE Transactions on Software Engineering*, SE-6(5):440–449, 1980.

[26] C. L. Seitz. "System Timing". In *Introduction to VLSI Systems*. by C. Mead and L. Conway, Addison Wesley, pages 128–262, 1980.

[27] E. A. Snow. Automation of Module Set Independent Register-Transfer Level Design. Technical Report Ph.D. Thesis, Electrical Engineering Department, Carnegie-Mellon University, April 1978.

[28] I. E. Sutherland. "MICROPIPELINES", The 1988 Turing Award Lecture. *Communications of the ACM*, 32(6):720–738, 1989.

[29] H. Trickey. "Flamel: A High-Level Hardware Compiler". *IEEE Transactions on Computer-Aided Design*, 6(2):259–269, March 1987.

[30] J. L. A. van de Snepscheut. "Deriving Circuits from Programs". *Third Caltech Conference on Very Large Scale Integration*, pages 241–256, March 1983.

[31] R. A. Walker and D. E. Thomas. "Design Representation and Transformation in The System Architect's Workbench". In *Proceedings of the ICCAD-87*, pages 166–169, 1987.

[32] T.-Y. Wuu. A Data-Driven Model for Asynchronous System Synthesis. Technical Report Thesis Proposal, Electrical Engineering-Systems, University of Southern California, December 1991.

[33] T.-Y. Wuu and S.B.K. Vrudhula. A Design of a Fast and Area Efficient Multi-input Muller C-element. *IEEE Transactions on VLSI Systems*, 1(2):215–219, 1993.

[34] *The HP C34100 Standard Cell Library Data Manual.* Hewlett-Packard Company, Integrated Circuit Business Division, May 1992.

[35] *Guidelines for Using The MOSIS Netlist-to-Parts Service, ViewLogic CMOSN Design Kit.* USC/ISI, MOSIS Project, October 1992.

[36] *The MOSIS Service.* USC/ISI, MOSIS Project, July 1993.

[37] *MOSIS prices and gate equivalents.* Tanner Research, April 1989.